Основы

C++

Методы программной инженерии



- Методы построения программ на C++
- Объектно-ориентированный подход к созданию программ
- Преимущества C++
- Создание легко сопровождаемых программ

Издательство "Лори"

Виктор Штерн

Отзывы

"Виктор Штерн раскрывает методы создания сопровождаемого ПО. Эта книга полезна программистам любого уровня. Она дает углубленное понимание предмета, легко читается, содержит ценные советы и полезные примеры".

Дэниэл Ф. Костелло,
ведущий специалист по разработке ПО
компании GE Marquette Medical Systems

"Великолепная книга и для начинающих программистов, и для тех, кто уже имеет опыт создания программ на языке C. Будет очень полезна тем, кто хочет получить четкое понимание концепций языка C++ и базовых принципов объектно-ориентированного программирования".

Стив Гласс,
ведущий специалист по разработке ПО
компании Motorola

CORE C++

A Software Engineering Approach

VICTOR SHTERN



Prentice Hall PTR, Upper Saddle River, NJ 07458
www.phptr.com

Основы C++

Методы программной инженерии

Виктор Штерн



Издательство "Лори"

CORE C++: a Software Engineering Approach
Victor Shtern
Copyright © 2000
All rights reserved

Основы C++: Методы программной инженерии
Виктор Штерн

Переводчик С. Орлов
Научный редактор А. Вендров
Корректоры Н. Литвинова, М. Ромашова
Верстка Т. Кирпичевой



Copyright © 2000 Prentice Hall PTR
Prentice-Hall, Inc.
Upper Saddle River, NJ 07458
World rights reserved.

ISBN 0-13-085729-7

© Издательство "Лори", 2003

Изд. № : OAI (03)
ЛР № : 070612 30.09.97 г.
ISBN 5-85582-188-9

Подписано в печать 20.01.2003 Формат 84 ×108/16
Бумага офсет № 1 Гарнитура Литературная Печать офсетная
Печ. л. 55 Тираж 3200 Заказ № 614
Цена договорная

Издательство "Лори". 123 100 Москва, Шмитовский пр., д. 13/6, стр. 1, (пом. ТАРП ЦАО)
Телефон для оптовых покупателей: (095)256-02-83
Размещение рекламы: (95)259-01-62

WWW.LORY–PRESS.RU

Отпечатано с диапозитивов в ЗАО "Красногорская типография".
143400, Моск. обл, г. Красногорск, Коммунальный квартал, д. 2

Посвящается Людмиле

Благодарности

Очень многие помогали мне в работе над этой книгой, и я хотел бы поблагодарить всех их. Прежде всего, я очень признателен Бьерну Страуструпу за разработку великолепного и мощного языка программирования. Мы все перед ним в долгу.

Далее я хочу поблагодарить Тимоти Бадда, Тома Каргилла, Джима Коплиена, Кэй Хорстманн, Айвора Хортона, Бертрана Мейера, Скотта Мейерса и Стивена Прата. Они написали книги по программированию, которые помогли мне выработать собственное представление о программировании на C++.

Кроме того, я в долгу перед своими слушателями в Бостонском университете, участниками профессиональных семинаров разработчиков и проводившихся на местах учебных курсах. Их вопросы и проблемы помогли мне понять, какие именно методы работают при изучении C++.

Я искренне благодарен сотрудникам Бостонского университета Тане (Стоянке) Златевой, Джею Халфонду и Дэннису Беркли за их веру в данный проект. Уделив мне время для завершения этой книги, они сделали возможным ее появление на свет.

Хотел бы отметить и работу редакторов Стива Гласса, Дэна Костелло и С. Л. Тондо. Благодаря им в этой книге удалось устранить немало досадных ошибок, которые я не заметил сам.

Я благодарен ответственному редактору издательства Prentice Hall Джиму Маркхаму за помощь и содействие в работе. Именно Джим первым отметил высокое качество данной книги, хотя английский — не мой родной язык. Кроме того, он не препятствовал моим "духовным поискам", но в то же время не позволял мне выбиться из плана. И почти преуспел в этом.

Выражаю благодарность ответственному редактору Prentice Hall Яну Шварцу и его сотрудникам-корректорам, которые терпеливо боролись с моим русским стилем артиклей, предлогов и сделали язык этой книги больше похожим на английский.

Но прежде всего я благодарен Тиму Муру, ответственному редактору Prentice Hall. Он нашел время, чтобы выслушать мои предложения, имел достаточно воображения, чтобы мне поверить, и энтузиазм, чтобы убедить всех, какая это будет великолепная книга. Если бы не Тим, данная книга никогда не была бы написана. Спасибо, Тим! Ваши усилия так много значили!

Спасибо моей семье. Мои родные поощряли и воодушевляли меня. Они позволили мне написать данный труд, *не отвлекая* чрезмерно на утомительные работы по дому. Последнее особенно важно: пока я писал свою книгу, моя жена трудилась над своими. Я также поощрял ее, восхищался ею и позволил ей заниматься своей работой, *помогая* в работе по дому.

Наконец, не могу не отметить сенатора Генри М. Джексона (округ Вашингтон) и члена Палаты представителей Чарльза Вэника (округ Огайо), авторов поправки Джексона-Вэника, увязавшей права человека с благоприятствующим режимом торговли с зарубежными странами и столь долго подвергавшейся критике. Я — один из немногих счастливцев, на жизнь которых эта поправка повлияла самым непосредственным образом. Разница между свободой и рабством — мой личный опыт, а не просто абстрактная концепция. И за это я очень благодарен.

Предисловие

Поздравляю! Перед вами одна из самых полезных книг по C++. В ней рассказано о сильных и слабых сторонах языка C++, и сделано это лучше, чем в каком-либо другом написанном на текущий день руководстве. А их выпущено уже немало.

Чем эта книга отличается от других книг по C++

Конечно, любой автор может заявить, что его книга — одна из лучших. Данную книгу отличает то, что она написана с точки зрения программной инженерии (software engineering) и сопровождения ПО на C++. Лишь в очень немногих руководствах используется такой подход (если они вообще есть). Дело в том, что язык C++ меняет не только способ *написания* компьютерных программ, но и методы *изучения* языка программирования. В добрые старые времена достаточно было потратить день-другой на ознакомление с основами базового синтаксиса языка — и можно пробовать свои силы на простых примерах. Затем — переход к более сложному синтаксису и решение задач посложнее. Через неделю или две (либо три-четыре, если это действительно трудный язык) вы осваивали все и считали себя экспертом.

С языком C++ все по-другому, он очень многосторонний и сложный. Конечно, он является надстройкой над языком C, а потому простые программы на C (и, следовательно, на C++) научиться писать нетрудно. А вот со сложными программами дело обстоит иначе. Если программист недостаточно хорошо знает C++, то такая программа не будет переносимой и легко сопровождаемой, а повторное использование программного кода окажется почти невозможным.

C++ — замечательный язык. Он задумывался как язык общего назначения, и его создателям удалось добиться очевидного успеха. Сегодня C++ выбирают для решения экономических и инженерных задач и даже для написания приложений реального времени. На создание данного языка были потрачены значительные усилия. Для достижения высокой производительности программ в C++ поддерживается динамическое управление памятью, а различные части программы могут быть относительно независимыми. Даже если программа на C++ будет корректной синтаксически и полностью протестированной, не исключены следующие проблемы:

1. Медленное выполнение — намного медленнее сопоставимой программы на языке C.
2. Ошибки управления памятью, проявляющиеся только при изменении режима использования памяти (например, при инсталляции другой программы). Эти ошибки могут приводить к аварийному завершению программы или просто к некорректным результатам.
3. Зависимость между отдельными частями программы, затрудняющая понимание замысла разработчика со стороны службы сопровождения. Плохо написанные программы на C++ намного труднее сопровождать и повторно использовать, чем необъектно-ориентированные программы.

Насколько все это важно? Если создается простая программа, которая прослужит непродолжительное время, то скорость ее выполнения, пригодность кода для повторного использования, управление памятью и сопровождаемость, возможно, и не так важны. Все, что требуется — быстрое решение проблемы. Если оно получилось неудовлетворительным, можно избежать потерь, написав вместо прежней новую программу. Для этого подойдет любая книга по C++ (но надо прочитать и эту, насладившись ее неформальным стилем, углубленным знакомством с языком и особенностями его использования).

Но для коллективной работы при создании больших приложений, которые придется сопровождать длительное время, все это имеет значение. Методы программной инженерии и сопровождения ПО, описываемые в данной книге, будут в таком случае весьма полезны. В большинстве книг по C++ подобные вопросы не затрагиваются вовсе. (Достаточно просмотреть их тематические указатели.) А если и затрагиваются, то в них не раскрываются методы, позволяющие выйти из трудной ситуации.

Еще одна важная черта этой книги — способ подачи материала. Есть немало книг, где перечисляются средства и особенности языка, но почти не рассказывается о том, как его использовать. Это все равно, что, изучая французский язык, заниматься только грамматикой. Можно ли, освоив грамматику, заговорить по-французски? Очевидно, что для беглой речи этого недостаточно. В данной книге описано,как нужно и как не нужно применять язык, особенно с точки зрения повторного использования и последующего сопровождения программ.

Особенностью C++ является тесная взаимосвязь различных средств языка, из-за чего его очень трудно осваивать последовательно — от простого к сложному. Многие авторы руководств по C++ даже и не пытаются этого делать. Они полагают, что это только запутывает читателя, а в результате упоминают в главе 3 концепцию, которая поясняется лишь в главе 8, оставляя вас в недоумении.

Здесь реализован другой подход. Темы раскрываются циклически — сначала дается общий обзор, затем более углубленный со всеми особенностями, а потому для понимания материала не потребуется перепрыгивать через главы.

Такой подход разрабатывался автором за долгие годы преподавания и обучения специалистов в области ПО. Большинство его учеников, слушателей колледжа при Бостонском университете (Boston University Metropolitan College), уже выполняли высококвалифицированную работу ранее, другие решили расширить свое образование с целью профессионального роста. Автор провел бесчисленное множество семинаров и учебных курсов, хорошо знаком со сложностями освоения учащимися принципов языка и методов программирования. Этот опыт позволил хорошо продумать последовательность изложения тем, примеры, контрпримеры и рекомендации. Такой подход к обучению C++ достаточно уникален, и читатель может воспользоваться его преимуществами.

Для кого предназначена книга

Книга написана для профессионалов, которым необходимо осмысленное, продуманное представление практических деталей программирования на C++ и понимание всех тонкостей языка.

Эта книга для тех, кто хочет познакомиться с практическим обсуждением новых технологий и методами их использования, а также для тех, кто уже знаком с другими языками и собирается перейти на C++. Опытные программисты найдут данную книгу полезной, на что-то она откроет им глаза. Читатели, для которых это первая книга по программированию (что даже хорошо), будут вознаграждены за усилия, потраченные на ее изучение.

Как построена эта книга

Вряд ли нужно подробно пояснять, о чем и где рассказывается. Незнакомые термины, концепции и методы здесь перечислять не стоит — это лишь утомит читателя. Именно поэтому краткое содержание книги вынесено в конец, в главу 19. Те, кого это интересует, могут сначала ее прочитать.

Лучше отметить, какие части представляют интерес для читателей с разным опытом программирования.

- Программистам, имеющим опыт работы с C++, наиболее интересны будут части III и IV, где рассказывается о мощных средствах C++ и "подводных камнях". Если при первом знакомстве с C++ вы поспешили начать работу с объектами, не освоив как следует методы процедурного программирования, управления памятью и создания удобного в сопровождении программного кода, то стоит ознакомиться также с частями I и II (кроме того, это просто интересно).
- Опытным программистам, применяющим язык C и желающим перейти на C++, следует прочитать части II, III и IV — это как раз для них. Полезно будет также просмотреть часть I, в которой язык C обсуждается с точки зрения разработки ПО и сопровождения программ.
- Программистам с опытом работы на языках высокого уровня, отличных от C и C++, следует начать с части I.
- Тем, кто хотел бы познакомиться с введением в программирование, лучше пропустить главу 1, посвященную объектно-ориентированному подходу. На этом этапе она будет для вас слишком абстрактной. Сначала изучите другие главы части I, потом вернитесь к главе 1, а затем переходите к частям II, III и IV.

Соглашения, используемые в данной книге

Весь программный код, представленный в нумерованных листингах, полностью протестирован и отлажен с использованием нескольких компиляторов, включая Microsoft Visual C++, Borland и GNU. Он должен работать без каких-либо изменений. Фрагменты кода, не включенные в нумерованные листинги, также отлажены и протестированы. Их можно выполнять, но для этого требуется некоторое знакомство с языком.

В данной книге листинги и фрагменты программного кода представлены моноширинным шрифтом. Это же относится к встречающимся в тексте терминам языка C++. Например, имя класса C++ записывается как `Account`, так же, как в программном коде. То же самое и с ключевыми словами.

Значками отмечены особенно полезные операторы или операторы, на которые стоит обратить внимание. Для этого же служат примечания, предупреждения и советы.



Внимание Данный значок поставлен рядом с информацией, заслуживающей особого внимания. Это может быть интересный факт, относящийся к освещаемой теме, или то, что читателю следует иметь в виду при написании программы.



Осторожно! Этим значком помечена информация о том, что может вызвать непредвиденные трудности или дать неожиданные результаты.



Советуем Особенно полезные сведения, помогающие читателям сэкономить время, ценные рекомендации по программированию или какие-либо конкретные советы для более продуктивной работы.

Доступ к исходному коду

В процессе изучения языка очень важно применять полученные знания на практике. Осваивать C++ без практики — все равно, что учиться вождению без автомобиля. Можно усвоить массу полезных сведений, но водить автомобиль так и не научиться. Автор настоятельно рекомендует поэкспериментировать с программами, обсуждаемыми в данной книге. Исходный код со всеми листингами можно найти на сайте:

```
ftp://ftp.prenhall.com/pub/ptr/c++programming.w-050/corec++
```

Обратная связь

Данная книга тщательно выверена и отредактирована. Тем не менее в ней могут содержаться некоторые ошибки.

Автор старался написать уникальную книгу, и, возможно, читатели встретят здесь утверждения, которые покажутся им безосновательными, неоправданными или просто ошибочными, противоречивыми и требующими обсуждения.

Если у вас будут какие-нибудь замечания или дополнения, пишите по адресу:

```
shtern@bu.edu
```

Содержание

Благодарности . . . . . *vi*
Предисловие . . . . . *vii*

Часть I
Введение в программирование на C++

Глава 1 Объектно-ориентированный подход: что это такое? 3

Истоки кризиса программного обеспечения 4
Выход первый: избавиться от программистов 7
Выход второй: совершенствование методов управления 9
Метод "водопада" 9
Быстрое прототипирование 10
Выход третий: разработка сложного и подробного языка 11
Объектно-ориентированный подход: что он дает и какой ценой? 12
Чем занимается проектировщик? 13
Качество проекта: сцепление 14
Качество проекта: связность 14
Качество проекта: связывание вместе данных и функций 15
Качество проекта: сокрытие информации и инкапсуляция 17
Проблемы проектирования: конфликты имен 18
Проблемы проектирования: инициализация объектов 18
Так что же такое объект? 19
Преимущества применения объектов 20
Характеристики языка программирования C++ 21
Цели языка C: производительность, удобочитаемость, красота и переносимость 21
Цели языка C++: классы и обратная совместимость с языком C 23
Итоги 25

Глава 2 Быстрый старт: краткий обзор C++ 27

Базовая структура программы 28
Директивы препроцессора 30
Комментарии 33
Описания и определения 36
Операторы и выражения 41
Функции и вызовы функций 47
Классы 55
Применение инструментальных средств разработки программного обеспечения 58
Итоги 62

Глава 3 Работа с данными и выражениями C++ 63

Значения и их типы . . . 63
Интегральные типы . . . 66
Спецификаторы типов . . . 67
Символы . . . 71
Булевы значения . . . 73
Типы с плавающей точкой . . . 73
Работа с выражениями C++ . . . 75
Высокоприоритетные операции . . . 76
Арифметические операции . . . 77
Операции сдвига . . . 79
Поразрядные логические операции . . . 80
Операции отношения и равенства . . . 82
Логические операции . . . 84
Операции присваивания . . . 85
Условная операция . . . 87
Операция запятой . . . 87
Смешанные выражения: скрытая опасность . . . 88
Итоги . . . 93

Глава 4 Управление ходом выполнения программы C++ 95

Операторы и выражения . . . 96
Условные операторы . . . 97
Стандартные формы условных операторов . . . 97
Распространенные ошибки в условных операторах . . . 101
Вложенные условные операторы и их оптимизация . . . 110
Итерация . . . 117
Применение цикла while . . . 117
Итерации в цикле do-while . . . 124
Итерации с циклом for . . . 127
Операторы перехода в C++ . . . 130
Оператор break . . . 131
Оператор continue . . . 133
Оператор goto . . . 134
Переходы return и exit . . . 135
Оператор switch . . . 139
Итоги . . . 142

Глава 5 Агрегирование с помощью типов данных, определяемых программистом 143

Массивы как однородные агрегаты . . . 144
Массивы как векторы значений . . . 144
Определение массивов в C++ . . . 145
Операции с массивами . . . 147
Проверка допустимости индекса . . . 148
Многомерные массивы . . . 151
Определение символьных массивов . . . 154
Операции с символьными массивами . . . 155
Строковые функции и порча содержимого памяти . . . 156
Двумерные символьные массивы . . . 160

Переполнение массива в алгоритме вставки . . . 162
Определение типов массивов . . . 165
Структуры как неоднородные агрегаты . . . 167
Структуры, определяемые программистом . . . 167
Создание и инициализация переменных-структур . . . 168
Иерархические структуры и их компоненты . . . 169
Операции с переменными-структурами . . . 170
Определение структур в многофайловых программах . . . 172
Объединения, перечисления и битовые поля . . . 173
Объединения . . . 173
Перечисления . . . 176
Битовые поля . . . 179
Итоги . . . 182

Глава 6 Управление памятью: стек и динамически распределяемая область 183

Область действия имени как средство кооперации . . . 184
Лексические области действия в C++ . . . 184
Конфликты имен в одной области действия . . . 185
Использование одинаковых имен в независимых областях действия . . . 188
Использование одинаковых имен во вложенных областях действия . . . 189
Область действия переменных цикла . . . 193
Управление памятью: классы памяти . . . 193
Автоматические переменные . . . 195
Внешние переменные . . . 197
Статические переменные . . . 202
Управление памятью: использование динамически распределяемой области 206
Указатели C++ как типизированные переменные . . . 207
Выделение памяти в динамической области . . . 212
Массивы и указатели . . . 215
Динамические массивы . . . 218
Динамические структуры . . . 230
Обмен данными с файлами на диске . . . 239
Вывод в файл . . . 239
Ввод из файла . . . 241
Файловые объекты ввода и вывода . . . 245
Итоги . . . 248

Часть II
Объектно-ориентированное программирование на C++

Глава 7 Программирование с использованием функций C++ 253

Функции C++ как средства разбиения программы на модули . . . 254
Объявление функции . . . 255
Определения функций . . . 256
Вызовы функций . . . 257
Преобразование типов аргументов . . . 258

Передача параметров в C++ 260
Передача по значению 260
Вызов по указателю 262
Передача параметров в C++ по ссылке 268
Структуры 271
Массивы 276
Еще о преобразовании типов 279
Возврат значения из функции 281
Встраиваемые функции 287
Параметры с заданными по умолчанию значениями 289
Перегрузка имен функций 293
Итоги 300

Глава 8 Объектно-ориентированное программирование с использованием функций 302

Сцепление 305
Связность 305
Неявная связность 306
Явная связность 309
Как уменьшить степень связности 313
Инкапсуляция данных 317
Сокрытие информации 322
Большой пример инкапсуляции 327
Недостатки инкапсуляции с использованием функций 335
Итоги 337

Глава 9 Классы C++ как единицы модульности программы 339

Базовый синтаксис класса 341
Связывание операций и данных 341
Исключение конфликтов имен 345
Реализация функций-членов вне класса 348
Определение объектов классов с разными классами памяти 351
Управление доступом к компонентам класса 352
Инициализация экземпляров объекта 357
Конструкторы как функции-члены 358
Конструкторы, используемые по умолчанию 360
Конструкторы копирования 362
Конструкторы преобразования 364
Деструкторы 366
Время вызова конструктора и деструктора 370
Область действия класса и подмена имен во вложенных областях 370
Управление памятью с помощью операций и вызовов функций 372
Использование в коде клиента возвращаемых объектов 376
Возврат указателей и ссылок 376
Возврат объектов 378
Еще о ключевом слове `const` 381
Статические компоненты класса 386
Применение глобальных переменных как характеристик класса 386
Четвертый смысл ключевого слова `static` 388

Инициализация статических элементов данных . . . 389
Статические функции-члены . . . 390
Итоги . . . 393

Глава 10 Операторные функции 394

Перегрузка операций . . . 395
Ограничения перегрузки операций . . . 402
Какие операции не могут быть перегруженными . . . 402
Ограничения на возвращаемые типы . . . 404
Ограничения на число параметров . . . 405
Ограничение на старшинство операций . . . 406
Перегруженные операции как компоненты класса . . . 406
Замена глобальной функции компонентом класса . . . 407
Использование членов класса для цепочек операций . . . 409
Применение ключевого слова const . . . 410
Учебный пример: рациональные числа . . . 412
Параметры смешанных типов . . . 420
Дружественные функции . . . 427
Итоги . . . 438

Глава 11 Конструкторы и деструкторы: потенциальные проблемы 440

Передача объектов по значению . . . 441
Перегрузка операций для нечисловых классов . . . 446
Класс String . . . 446
Динамическое управление памятью . . . 448
Защита данных объекта от клиента . . . 451
Перегруженная операция конкатенации . . . 451
Предотвращение "утечек памяти" . . . 453
Защита целостности программы . . . 454
Переход из пункта A в пункт B . . . 458
Конструктор копирования . . . 459
Решение проблем целостности . . . 460
Семантика копирования и семантика значений . . . 464
Конструктор копирования, определяемый программистом . . . 465
Возврат по значению . . . 469
Ограничения для эффективности конструктора копирования . . . 472
Перегрузка операции присваивания . . . 472
Проблемы системной операции присваивания . . . 473
Перегруженная операция присваивания: первая версия (утечка памяти) . . . 473
Перегруженная операция присваивания: следующая версия (самоприсваивание) . . . 475
Перегруженная операция присваивания: еще одна версия (цепочка выражений) . . . 475
Вопросы производительности . . . 478
Первое решение: больше перегруженных операций . . . 479
Второе решение: возврат по ссылке . . . 480
Практические вопросы: что подлежит реализации . . . 481
Итоги . . . 484

Часть III
Объектно-ориентированное программирование с агрегированием и наследованием

Глава 12 Преимущества и недостатки составных классов 487

Использование объектов классов как элементов данных 488
Синтаксис C++ для композиции класса 490
Доступ к элементам данных элементов данных класса 492
Доступ к элементам данных параметров метода 494
Инициализация составных объектов 495
Применение используемых по умолчанию конструкторов компонента 497
Использование списка инициализации элементов 502
Элементы данных со специальными свойствами 507
Константы как элементы данных 507
Ссылочные элементы данных 508
Использование объектов как элементов данных своего собственного класса 511
Использование статических элементов данных как компонентов собственного класса 513
Контейнерные классы 515
Вложенные классы 529
"Дружественные" классы 531
Итоги 534

Глава 13 Подобные классы и их интерпретация 535

Интерпретация подобных классов 537
Слияние свойств подклассов в один класс 538
Перенос ответственности за целостность программы на сервер 539
Отдельные классы для каждого серверного объекта 544
Применение наследования C++ для связывания родственных классов 546
Синтаксис наследования в C++ 549
Различные режимы создания производного класса из базового класса 550
Определение и использование объектов базовых и производных классов 553
Доступ к сервисам базового и производного классов 555
Доступ к базовым компонентам объекта производного класса 558
Наследование компонентов `public` 559
Наследование в режиме `protected` 563
Наследование в режиме `private` 567
Изменение доступа к компонентам базового класса в производном классе 569
Режим наследования по умолчанию 570
Правила области действия и разрешение имен при наследовании 572
Перегрузка и сокрытие имен 574
Вызов метода базового класса, скрытого производным классом 578
Применение наследования для развития программы 581
Конструкторы и деструкторы для производных классов 584
Использование в конструкторах производных классов списков инициализации 587
Деструкторы при наследовании 590
Итоги 592

Глава 14 Выбор между наследованием и композицией 593

Выбор методики повторного использования кода · · ·

Множественное наследование: неоднозначность . . . 704
Множественное наследование: ориентированный граф . . . 706
Полезно ли множественное наследование . . . 708
Итоги . . . 709

Глава 16 Расширенное использование перегрузки операций 711

Перегрузка операций: краткий обзор . . . 711
Унарные операции . . . 719
Операции инкремента и декремента . . . 719
Постфиксные перегруженные операции . . . 727
Операции преобразования . . . 729
Операции, возвращающие компонент массива по индексу, и операции вызова функции . . . 736
Операции, возвращающие компонент массива по индексу . . . 737
Операция вызова функции . . . 745
Операции ввода/вывода . . . 750
Перегрузка операции >> . . . 750
Перегруженная операция << . . . 754
Итоги . . . 756

Глава 17 Шаблоны как еще одно средство проектирования 757

Простой пример повторного использования структуры класса . . . 758
Синтаксис определения шаблонного класса . . . 766
Спецификация шаблонного класса . . . 766
Реализация шаблона . . . 768
Реализация шаблонных функций . . . 769
Вложенные шаблоны . . . 775
Шаблонные классы с несколькими параметрами . . . 776
Несколько параметров типов . . . 777
Шаблоны с параметрами — константные выражения . . . 779
Взаимосвязи между реализациями шаблонных классов . . . 782
Шаблонные классы как "друзья" . . . 783
Вложенные шаблонные классы . . . 786
Шаблоны со статическими компонентами . . . 788
Специализации шаблонов . . . 790
Шаблонные функции . . . 794
Итоги . . . 796

Глава 18 Программирование с обработкой исключительных ситуаций 797

Простой пример для обработки исключительных ситуаций . . . 798
Синтаксис исключительных ситуаций C++ . . . 804
Генерация исключительной ситуации . . . 805
Отслеживание исключительной ситуации . . . 806
Обозначение исключительной ситуации . . . 812
Повторная генерация исключительной ситуации . . . 815
Исключительные ситуации с объектами класса . . . 817
Синтаксис объектов генерации, обозначения и отслеживания . . . 818
Использование наследования с исключительными ситуациями . . . 821
Стандартная библиотека исключительных ситуаций . . . 825

Операции приведения типов 827
Операция static_cast 827
Операция reinterpret_cast 831
Операция const_cast 831
Операция dynamic_cast 834
Операция typeid 836
Итоги 837

Глава 19 Полученные уроки 839

C++ как традиционный язык программирования 840
Встроенные типы данных C++ 840
Выражения C++ 842
Управляющая логика в C++ 843
C++ как модульный язык 844
Составные типы C++: массивы 844
Составные типы C++: структуры, объединения, перечисления 845
Функции C++ как средства обеспечения модульности 846
Функции C++: передача параметров 848
Область видимости и класс памяти в C++ 849
C++ как объектно-ориентированный язык 850
Классы C++ 850
Конструкторы, деструкторы и перегруженные операции 851
Композиция классов и наследование 853
Виртуальные функции и абстрактные классы 854
Шаблоны 855
Исключительные ситуации 856
C++ и его конкуренты 857
C++ и старшие языки 857
C++ и Visual Basic 857
C++ и C 858
C++ и Java 859
Итоги 860

Часть I

Введение в программирование на C++

Первая часть этой книги посвящена основам программирования на C++. Как известно, C++ — объектно-ориентированный язык программирования. Но что это означает? Для чего используется объектно-ориентированный язык? Чем он лучше традиционного, не объектно-ориентированного? На что нужно обращать внимание при программировании, чтобы реализовать преимущества объектно-ориентированного подхода (ООП)? Часто такой подход воспринимается как нечто само собой разумеющееся, что снижает эффективность его использования.

Ответы на данные вопросы даются в первой главе. В ней рассказывается, как разбивать программу на части. Большую программу можно написать как набор относительно независимых, но взаимодействующих друг с другом и совместно работающих компонентов. Если же мы разделим то, что должно быть объединено, это приведет к избыточным связям и зависимостям между частями программы, затруднит ее повторное использование и сопровождение. Если же сохранить как единый фрагмент блок, который следовало бы разделить на части, то увеличится сложность программного кода — в нем невозможно будет разобраться, и усложнится повторное использование и сопровождение ПО.

В использовании объектов нет ничего таинственного, но просто применение объектов само по себе преимуществ не даст. Если же продумать с точки зрения объектов всю программу, это позволит избежать двух опасностей: ее излишней фрагментации на блоки и совмещения того, что должно быть разделено. В главе 1 показано, какие проблемы следует решать с помощью ООП и в чем именно способен помочь подобный подход.

В главе 2 содержится краткое введение в язык C++, включая описания объектов. Оно дает представление о предмете в общих чертах. Детали читатели найдут в других главах. Тем не менее главы 2 вполне достаточно для того, чтобы научиться писать простые программы на C++, подготовиться к более подробному знакомству с сильными и слабыми сторонами C++.

В других главах части I представлены базовые не объектно-ориентированные средства языка. Особое внимание уделяется написанию повторно используемого и удобного в сопровождении программного кода. Для каждой конструкции C++ поясняется, как ее следует и как не следует применять. Хотя об объектах речи еще не идет, эти главы уже достаточно сложны, особенно глава 6, рассказывающая об управлении памятью. Не удивительно: C++ — действительно сложный язык. Пропустите не вполне понятные темы — к ним можно будет вернуться позднее, когда наступит время сосредоточиться на деталях программирования.

Глава 1

Объектно-ориентированный подход: что это такое?

Темы данной главы

- ✔ Истоки кризиса программного обеспечения
- ✔ Выход первый: избавиться от программистов
- ✔ Выход второй: усовершенствование методов управления
- ✔ Выход третий: разработка сложного и подробного языка
- ✔ Объектно-ориентированный подход: что он дает и какой ценой?
- ✔ Характеристики языка программирования C++
- ✔ Итоги

Объектно-ориентированный подход захватывает все области разработки программного обеспечения. Он открывает новые горизонты и дает новые преимущества. Многие разработчики воспринимают эти преимущества как нечто само собой разумеющееся, считают их важными и существенными. Но что это такое на самом деле? Достигаются ли они автоматически, лишь потому, что в программе применяются объекты, а не функции?

В данной главе сначала рассказывается, для чего нужен объектно-ориентированный подход. Опытные разработчики ПО могут пропустить это описание и перейти непосредственно к пояснению достоинств ООП.

Тем же, у кого относительно мало опыта, лучше познакомиться с обсуждением кризиса ПО и способов выхода из него. Это поможет лучше понять, в каком контексте следует воспринимать предлагаемые в данной книге методы программирования. Читатели узнают, какие приемы программирования на C++ повышают качество программы, а какие наоборот и почему.

Учитывая обилие некачественного программного кода на C++, это очень важно. Многие программисты полагают, что одно лишь применение C++ и его классов дает все связанные с ними преимущества. Это не так. К сожалению, в большинстве книг по C++ поддерживается данное некорректное утверждение и подобное восприятие C++, а все внимание концентрируется на синтаксисе языка. Авторы предпочитают не обсуждать качество программного кода на C++, но если разработчики не знают, каковы цели C++, они станут писать объектно-ориентированные программы по-старому. Подобные программы будут не лучше, чем традиционные программы на C, PL/I (или другом подобном языке), а их трудно сопровождать.

Истоки кризиса программного обеспечения

Объектно-ориентированный подход — это еще одна попытка отрасли преодолеть так называемый кризис ПО, на который указывают частые перерасходы бюджета, срывы сроков проектов или отказ от их реализации, недостаточная функциональность систем и множество ошибки в ПО. Негативные последствия ошибок в ПО могут быть разными — от простого неудобства для пользователей до серьезных экономических убытков из-за некорректно проведенной операции. К сожалению, из-за ошибок в ПО могут подвергаться опасности человеческие жизни и срываться важные проекты. Исправление ошибок обходится дорого и нередко ведет к резкому росту расходов на ПО.

Многие эксперты полагают, что причиной кризиса ПО является отсутствие стандартной методологии: отрасль еще слишком молода. Другие технические дисциплины гораздо более зрелые и уже имеют проверенные методы, методологии и стандарты.

Возьмем, к примеру, строительство. Стандарты, нормы и правила здесь используются очень широко. Каждый этап строительного процесса предусматривает детальные инструкции. Все участники знают, чего можно ожидать, и как продемонстрировать достижение того или иного критерия качества. Существуют определенные гарантии и средства проверки. Закон защиты прав потребителей охраняет их от недобросовестных или неумелых разработчиков и исполнителей.

То же самое в более новых отраслях — автомобильной или электротехнической. И здесь действуют отраслевые стандарты, общепринятые методы разработки и производства, существуют гарантии производителей и законы по охране прав потребителей. Еще одной важной характеристикой этих отраслей является сборка изделий из уже готовых компонентов — массово выпускаемых, стандартизированных и полностью протестированных.

Сравним все это с индустрией разработки ПО. О стандартах здесь речи нет. Конечно, профессиональные организации стараются сделать все возможное и выпускают различные стандарты — от спецификаций тестирования ПО до пользовательских интерфейсов. Но это лишь вершина айсберга. Универсальные, общепринятые, соблюдаемые всеми и обязательные методологии и процессы разработки ПО, увы, отсутствуют. Потребителю повезет, если производитель несет ответственность за стоимость носителя, на котором распространяется программный продукт. Возврат товара не практикуется: открыв коробку, можно позабыть о своих правах и возможности вернуть назад деньги.

Все продукты — ручной сборки. Готовых, стандартных компонентов нет. Нет и универсального общепринятого соглашения по таким продуктам и компонентам. В судебном деле правительства США против Microsoft обвинение и защита долго спорили о том, что такое *операционная система* и ее компоненты: можно ли считать браузер частью ОС или это просто еще одно приложение типа текстового редактора и электронной таблицы. Операционная система столь же важна для компьютера, как система зажигания для автомобиля (и даже важнее). Но можно ли подвергать сомнению то, из каких компонентов состоит система зажигания? Известно, что, когда этого требовала технология, карбюратор был частью системы зажигания. После изменения технологии он перестал входить в эту систему. Все обошлось без публичных споров.

Новизна индустрии ПО, конечно, сказывается на ситуации. Остается надеяться, что некоторые элементы этой мрачной картины в будущем исчезнут. Между тем, молодость не помешала данной отрасли превратиться в индустрию с многомиллиардными оборотами, играющую решающую роль в экономике. Интернет изменил методы ведения коммерции и поиска информации. Неузнаваемо изменил он и биржевой рынок.

Еще не так давно Проблема 2000 г. считалась большой угрозой для экономики. Не важно, оправдались эти опасения или нет. Важнее то, что индустрия ПО созрела достаточно, чтобы продемонстрировать свою мощь. Если проблема с ПО способна подорвать сами устои общества — значит, данная отрасль играет весомую роль. Между тем ее технология отстает от технологии других отраслей, и в основном это связано с процессом разработки ПО.

Лишь немногие программные системы настолько просты, что спецификацию для них может разработать один человек, реализовать их по данной спецификации и использовать для задуманной цели, а также сопровождать, внося изменения при появлении новых требований или обнаружении ошибок. Подобные простые системы достаточно ограничены по своему применению и живут относительно недолго. При необходимости их можно просто выбросить и начать все сначала. Инвестиции (время/деньги) невелики, и программы нетрудно переписать заново.

Но большинство программ имеют совсем другие характеристики. Это сложные системы, которые не под силу реализовать одному человеку. В их создании участвует целый коллектив, и разработчики должны координировать свою деятельность. Когда задача распределяется по нескольким исполнителям, имеет смысл сделать части программы, за которые они отвечают, относительно независимыми. Это облегчит индивидуальную работу программистов.

Например, можно разбить функции системы на отдельные операции (размещение заказа, добавление заказчика, удаление заказчика и т. д.). Если эти операции слишком сложны, их реализация с помощью отдельной программы потребует очень много времени, поэтому имеет смысл разбить каждую операцию на шаги и подшаги (такие, как проверка заказчика, ввод данных заказа, анализ кредитоспособности заказчика и пр.) и дать задание на их реализацию отдельным программистам (см. рис. 1.1).

Цель состоит в том, чтобы сделать компоненты системы независимыми друг от друга, благодаря чему программисты смогут работать над ними индивидуально. Между тем на практике эти отдельные фрагменты независимыми не являются. Кроме всего прочего, это части одной системы, а потому программы должны вызывать друг друга, работать с общими структурами данных или реализовывать различные шаги одного алгоритма. Так как фрагменты, написанные разными людьми, не являются независимыми, разработчикам приходится координировать и кооперировать свои усилия. Они пишут различные памятки, проектные документы, обмениваются сообщениями электронной почты, участвуют в совещаниях, анализе проекта и программного кода. Вот здесь и возникают ошибки: кто-то что-то не так понял, пропустил или не изменил, когда было принято соответствующее решение.

Проектирование, разработка и тестирование подобных сложных систем занимают очень много времени, обходятся дорого и влияют на работу большого числа пользователей. При изменении или появлении новых требований, обнаружении



Рис. 1.1. *Разбиение системы на компоненты*

ошибок такие системы нельзя просто заменить или выбросить — слишком много средств в них вложено.

Подобные системы должны быть сопровождаемыми, а их программный код — изменяемым. Изменения, внесенные в одном месте программы, нередко влияют на другие ее части, т. е. влекут за собой очередные изменения. Если такие зависимости не отмечены (а иногда просто *пропущены*), то система будет работать некорректно, пока ее программный код не изменят снова (что опять может повлечь необходимость изменений в остальных частях программы). Поскольку с этими сложными системами обычно связаны значительные инвестиции, они сопровождаются в течение долгого времени, хотя сопровождение программного кода сложных систем также обходится дорого и может стать источником новых ошибок.

Здесь также уместно вспомнить о Проблеме 2000 г. Многих удивлял тот факт, что для представления года программисты использовали только две цифры. "В каком мире они живут,— интересовалась публика.— Неужели не понимают, какие последствия повлечет изменение даты с 1999 на 2000 г.?" Да, это и впрямь удивительно, но удивляет вовсе не недальновидность программистов, а срок жизни систем, созданных в 70-е и 80-е годы. Программисты понимали проблему 2000-го не хуже любого эксперта по Y2K (или даже лучше), но они и думать не могли, что кто-то будет использовать их программы через 20—30 лет.

Да, сегодня многие организации тратят немало средств на сопровождение ПО — как будто хотят перещеголять в этом других. Подобные системы настолько сложны, что перестраивать их заново намного дороже, чем сопровождать.

Сложность — одна из важных характеристик большинства систем ПО. Сложны сами решаемые проблемы, управление процессом разработки, а вот методы создания ПО из отдельных фрагментов не соответствуют всей этой сложности.

Сложность системных задач (проблемной области), будь то инженерная задача, деловые операции, Интернет-приложение или готовое ПО для массового рынка, затрудняет описание системы и реализуемых ею функций для конечного пользователя. Потенциальные пользователи системы (или специалисты по маркетингу) не всегда в состоянии выразить свои требования в форме, понятной разработчикам ПО. Требования часто исходят от пользователей разных категорий и противоречат друг другу. Выявление и согласование всех расхождений — непростая задача. Кроме того, потребности пользователей и маркетологов со временем меняются, а иногда это происходит уже на этапе формулирования требований, и тогда в ходе обсуждения деталей, относящихся к работе системы, рождаются новые идеи. Вот почему программисты говорят, что пользователи (и специалисты по маркетингу) не знают, чего хотят. Инструментальных средств для работы с предъявляемыми к системе требованиями очень мало, а потому на выходе нередко получаются целые тома текста с чертежами. Этот текст часто плохо структурирован, и разобраться в нем нелегко. Многие положения весьма туманны, неполны, противоречивы и не поддаются однозначной интерпретации.

Сложность управления процессом разработки проистекает из необходимости координировать деятельность большого числа специалистов, особенно когда коллектив, работающий над разными частями проекта, разбросан географически и приходится согласовывать компоненты системы или процедуры работы с данными, например, в одной части системы данные выражены в ярдах, а в другой — в метрах. Подобное согласование просто по своей сути, но не по объемам, а удержать все в голове бывает очень сложно. Увеличение числа работающих над проектом сотрудников помогает не всегда. Новым сотрудникам приходится вникать в задачи, на это уходит какое-то время. Обычно новичкам перепадает часть задачи, над которой, как предполагалось, другие сотрудники будут работать позднее, или проект делится на все более мелкие куски.

Продуктивной работы от вновь включенных в проект сотрудников сразу ожидать не приходится. Сначала им нужно познакомиться с решениями, уже принятыми другими программистами. Это замедляет весь процесс, отвлекая внимание занятых в нем сотрудников на обучение новых.



Рис. 1.2. *Магический треугольник программных проектов*

Создание ПО из отдельных имеющихся компонентов только добавляет проблем: это требует времени и порождает новые ошибки. Тестирование таких систем затруднительно, оно ненадежно и требует "ручных" операций.

Когда я приехал в США, мой босс Джон Конвей пояснил ситуацию следующим образом. Он нарисовал треугольник, вершины которого представляют такие характеристики проекта, как сроки, бюджет и функциональность системы (рис. 1.2). "Мы не можем добиться всех трех целей,— сказал он.— Если реализовать все функции системы согласно бюджету, то работу невозможно будет завершить вовремя, потребуется перенос сроков. Если же реализовать функции в соответствии со сроками, то, скорее всего, бюджет придется перерасходовать понадобятся дополнительные ресурсы. Когда соблюдаются сроки и бюджет (что случается нечасто, но возможно), то придется пожертвовать некоторыми функциями и воплотить только часть обещанного".

С этой проблемой уже давно сталкивается индустрия ПО. Впервые о кризисе ПО заговорили в 1968 г. В последующие годы отрасль разработала несколько подходов к решению проблемы. Ниже мы вкратце рассмотрим их.

Выход первый: избавиться от программистов

В прошлом стоимость аппаратного обеспечения преобладала над стоимостью программного обеспечения. Программные продукты были относительно дешевы. "Узким" местом в процессе разработки системы казалось взаимодействие между программистами и пользователями ПО, которые пытались объяснить программистам, что же нужно для решения их коммерческих или технических задач.

С одной стороны, программисты не могли уяснить сути дела, поскольку имели математическую подготовку, были несведущи в бизнесе, технических областях и т. д., не знали соответствующей терминологии. С другой стороны, специалисты в бизнесе и технических областях были незнакомы с терминологией программирования и спецификой разработки ПО, а потому, когда программисты пытались уяснить для себя требования, взаимопонимания с пользователями не получалось.

Аналогично программисты нередко не понимали целей пользователей, их предположений и ограничений. В результате заказчики ПО получали не совсем то, что они хотели.

Тогда хорошим выходом из кризиса казалось кардинальное решение проблемы: избавиться от программистов. Пусть специалисты по бизнесу и инженеры сами пишут приложения, не обращаясь к посредникам-программистам. В те времена программисты использовали машинные языки и ассемблер. Эти языки требовали хорошего знания архитектуры компьютеров и набора команд, слишком трудного для тех, кто не имел специальной подготовки.

Чтобы реализовать задуманное решение, нужен был язык программирования, позволяющий создавать ПО намного быстрее и проще. Такие языки должны быть удобными в использовании, чтобы инженеры, ученые и менеджеры могли писать программы самостоятельно, не поясняя программистам, что должно получиться в результате.

В качестве таких языков были предложены Фортран и Кобол. Подход оправдал себя. Многие ученые, инженеры и менеджеры действительно научились писать программы, что дало некоторым экспертам основание предрекать скорое исчезновение профессии программиста.

Между тем это сработало только для небольших программ, которые можно описать, спроектировать, реализовать и сопровождать в одиночку. Подход оправдал себя для программ, не требующих при их создании кооперации нескольких разработчиков и не рассчитанных на многолетнее сопровождение. Для написания небольших программ не было нужды в согласовании разных частей программы.

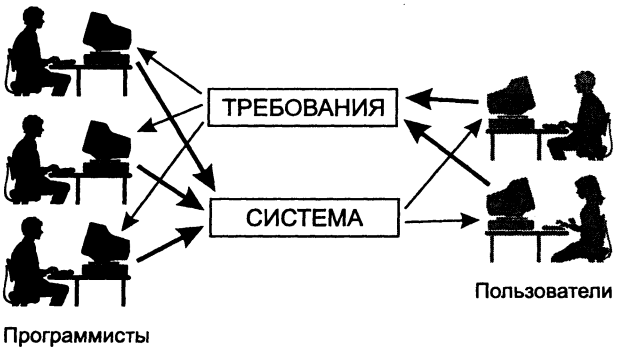


Рис. 1.4. *Схема взаимодействия разработчика и пользователя*

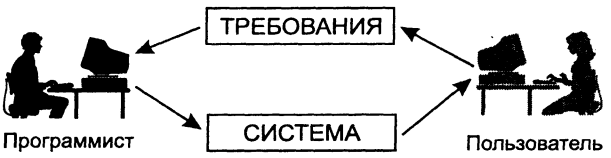


Рис. 1.3. *Схема взаимодействия пользователя и разработчика*

Фактически схема, приведенная на рис. 1.3, справедлива только для небольших программ. Для более крупных программных проектов картина будет выглядеть так, как показано на рис. 1.4. Проблемы взаимопонимания между пользователями и разработчиками действительно важны, но не менее важно и взаимопонимание между самими разработчиками. Если что-то будет понято не так, то, кем бы ни были разработчики — профессиональными программистами, инженерами или менеджерами,— неизбежны ошибки.

Даже рис. 1.4 дает упрощенную картину. Здесь показано лишь несколько пользователей, специфицирующих требования и оценивающих эффективность системы. У большинства программных проектов пользователей намного больше. Это касается как спецификаций (в которых могут участвовать представители отделов маркетинга и продаж), так и оценки (в данном процессе нередко участвует несколько человек). Несогласованность и пробелы в спецификациях, определяющих, что должна делать система (и оценках, насколько хорошо она это делает), добавляют проблемы в процессы взаимодействия разработчиков, особенно если система должна выполнять какие-то функции аналогично существующей системе. Разные разработчики нередко интерпретируют это по-разному.

Другая попытка избавиться от программистов основывалась на идее использования *суперпрограммистов*. Сама мысль проста. Если обычные программисты не могут создать части программы, работающие друг с другом без ошибок, то можно найти способного специалиста, настолько грамотного, чтобы разработать программу самостоятельно. Естественно, зарплата суперпрограммиста должна быть выше, чем зарплата программиста обычного. Если один человек создает разные части программы, то проблем с совместимостью меньше и устранить их можно быстрее.

Реально суперпрограммисты, конечно, не могли работать в одиночку — утомительные рутинные операции можно было передать обычным программистам с меньшим заработком. Таким образом, суперпрограммистов должны были поддерживать техники, библиотекари, тестировщики, технические писатели и т. д.

Такой подход имел ограниченный успех. Несмотря на его некоторые преимущества (в смысле сроков, бюджета и функций, вопреки пессимистической схеме рис. 1.2), коммуникации между суперпрограммистом и командой поддержки ограничивались обычными человеческими способностями этой службы поддержки.

Кроме того, суперпрограммисты не могли осуществлять долгосрочное сопровождение — они либо приступали к другим проектам, либо переходили на руководящую работу и переставали заниматься собственно программированием. Когда обычные программисты приступали к сопровождению созданного суперпрограммистами кода, они сталкивались с теми же или даже большими трудностями, поскольку суперпрограммисты оставляли после себя слишком бедную документацию: для них даже сложная система была относительно проста, и они не желали тратить время на ее подробное описание.

Сегодня лишь немногие пишут системы ПО без программистов. Отрасль перешла к поиску методов, позволяющих получить высококачественные программы с помощью людей обычных способностей. И она нашла выход в совершенствовании методов управления проектами.

Выход второй: совершенствование методов управления

Поскольку аппаратное обеспечение продолжает дешеветь, значительную часть в стоимости компьютерных систем составляет именно разработка и сопровождение ПО. Дорогое ПО — это существенные инвестиции. От такого ПО нельзя просто отказаться и переписать его заново. Следовательно, дорогие системы обслуживаются дольше, даже если это обходится недешево.

Растущая мощность аппаратного обеспечения открывает новые горизонты, но влечет за собой еще большее усложнение программного кода и стоимости ПО (как его разработки, так и сопровождения).

Это меняет приоритеты в процессе разработки ПО. Поскольку надежды решить данную проблему с помощью нескольких ярких индивидуальностей уже не вызывают такого энтузиазма, отрасль обратилась к методам управления взаимодействием между пользователями и разработчиками, в особенности — между разработчиками, отвечающими за разные части проекта.

Чтобы содействовать взаимодействию между пользователями и разработчиками, применяются следующие два метода:

- *Метод "водопада"* (waterfall) — разделение процесса разработки на отдельные стадии)
- *Быстрое прототипирование* —(частичная реализация системы для получения отзывов пользователей)

Метод "водопада"

Есть несколько вариантов такого подхода к управлению программными проектами. Все они включают в себя разбиение процесса на последовательные стадии. Типичная последовательность стадий такова: определение требований, анализ системы, проектирование архитектуры, детальное проектирование, реализация, тестирование модулей, интеграционное тестирование, приемочное тестирование и сопровождение. Обычно каждую стадию реализует отдельная команда разработчиков. После опытной эксплуатации и анализа полезности системы составляется новый набор требований (или поправки) и последовательность стадий может повторяться.

Переходы между этапами отражаются в плане проекта, где указываются сроки выпуска соответствующих документов. В идеальном случае разработанные на каждой стадии документы используются для двух целей: для обратной связи с разработчиками, отвечавшими за предыдущую стадию, и для оценки корректности решений/исходной документации следующей стадии проекта. Это можно делать неформально, путем циркулярной рассылки документа заинтересованным сторонам, или формально — с анализом стадий проекта и проведением совещаний с представителями каждой команды разработчиков и пользователями.

Например, в процессе определения требований создается документ с их перечислением, используемый для получения отзывов от инициаторов проекта или представителей заказчика и как исходный документ для системных аналитиков. Аналогично системные аналитики дают детальные спецификации системы для получения отзывов пользователей и составления исходного документа стадии проектирования. Но это в идеале. На практике люди, отвечающие за получение отзывов ("обратную связь"), могут иметь другие довлеющие над ними обязанности,

выделяя на обратную связь только часть времени. Это ставит под вопрос всю идею контроля качества в рамках процесса.

Кроме того, чем дальше продвигается реализация проекта, тем труднее получить от пользователей осмысленные отзывы. В терминологии им ориентироваться все сложнее (она становится более компьютерно-ориентированной), а в схемах и диаграммах применяются незнакомые пользователям обозначения. Анализ проекта часто делается формально, "под копирку".

Преимущество такого подхода — в хорошо спроектированной структуре с четко определенными для каждого разработчика ролями, конкретными требованиями (что должно получиться на выходе по завершении конкретной стадии) и сроками. Для планирования проекта, оценки его сроков и стоимости различных стадий предлагается ряд инструментальных средств. Это особенно важно для крупных проектов, когда нужно гарантировать, что проект движется в верном направлении. Опыт, накопленный при реализации одного проекта, помогает планировать следующие аналогичные проекты.

Недостаток — излишний формализм, замена персональной ответственности на групповую, неэффективность и задержка получения отзывов.

Быстрое прототипирование

В этом методе предпринимается противоположный подход, исключающий формальные стадии в пользу получения отзывов от пользователей. Вместо формальной спецификации системы создается ее прототип, опытный образец, который можно продемонстрировать пользователям. Пользователи испытывают этот прототип на гораздо более ранней стадии, чем в методе "водопада". Звучит неплохо, но для крупной системы проделать такое нелегко. Быстрое создание прототипа окажется вовсе не быстрым, а по сложности и стоимости может приближаться к стоимости реализации всей системы. Пользователи, получающие прототип на пробу, чаще озадачены своими прямыми обязанностями. Им не всегда хватает квалификации или времени для систематического тестирования и обратной связи с разработчиками.

Данный подход наиболее эффективен для определения пользовательского интерфейса системы: меню, диалоговых окон, текстовых полей, командных кнопок и других элементов взаимодействия между человеком и компьютером. Нередко организации пытаются комбинировать оба подхода. Это неплохо срабатывает, но не устраняет проблему взаимодействия между разработчиками, отвечающими за разные части системы.

Чтобы улучшить это взаимодействие, был предложен ряд формальных структурных методов, опробованных с той или иной степенью успеха. Для составления специальных требований и спецификаций может использоваться структурированный английский язык (или тот язык, на котором говорят разработчики), способствующий пониманию описания задачи и идентификации ее отдельных частей. Для определения общей архитектуры и специфических компонентов системы популярно сочетание структурного проектирования с такими методами, как диаграммы потоков данных и диаграммы переходов состояний. Для проектирования на низком уровне применяются различные виды блок-схем и структурированного псевдокода. Они способствуют пониманию алгоритмов и взаимосвязей между частями программы. Для их реализации применяются принципы структурного программирования. В нем ограничивается использование переходов в программе. Структурное программирование значительно облегчает понимание программного кода (по крайней мере, уменьшается сложность исходного кода программы).

Нет нужды описывать здесь каждый из этих методов. Подобные формальные методы управления и документирования могут быть очень полезными. Без них ситуация была бы хуже. Между тем они не избавляют от кризиса. Компоненты программ, которые затем соединяются с помощью взаимосвязей и зависимостей,

все равно создаются вручную. Разработчики сталкиваются с трудностями при документировании этих взаимосвязей, необходимом для того, чтобы работающие над другими частями системы поняли, чего от них ждут, и знали обо всех ограничениях. У программистов, занимающихся сопровождением ПО, также возникают трудности понимания сложных (и плохо документированных) взаимосвязей.

В результате отрасль обратилась к методам, позволяющим свести к минимуму последствия от неувязок в системе. В настоящее время происходит отход от методологий, ускоряющих и облегчающих написание ПО, к методологиям, упрощающим написание понятного ПО. Это не парадокс — такой сдвиг означает перенос внимания на качество программных продуктов.

Выход третий: разработка сложного и подробного языка

Раньше языки программирования (Фортран, Кобол, АПЛ, Бейсик и даже С) разрабатывались для упрощения написания программного кода. Эти языки программирования были относительно компактными, сжатыми и простыми в изучении. Считалось, что писать на них могут специалисты, имеющие самый незначительный опыт программирования. Позднее в разработке языков программирования произошел четкий сдвиг. В современных языках, включая C, Ada и Java, применяется иной подход. Они далеко не компактны и достаточно сложны в изучении. Написанные на этих языках программы длиннее аналогичных программ на традиционных языках. На программистов свалилось бремя определений, описаний и других вспомогательных элементов кода.

Такая "многословность" языков повлияла и на согласованность программного кода. Если программист использует в разных частях программы несогласованный код, компилятор обнаруживает это и вынуждает его устранить данную несогласованность. В прежних языках компилятор предполагал, что согласованность вводится преднамеренно для достижения некоей известной программисту цели. Разработчики языков и компиляторов считали: "Зачем думать за программиста?". В таких допускающих ошибки языках поиск ошибок часто требовал достаточно сложного тестирования на этапе выполнения, а некоторые ошибки все равно оставались невыловленными. В современных языках несогласованность кода интерпретируется как синтаксическая ошибка, выявляемая еще до выполнения программы. Это важное преимущество, но оно затрудняет написание программы.

Еще одно достоинство "многословности" языка — возможность лучше выразить намерения программиста. В традиционных языках программисту, занимающемуся сопровождением программного кода, часто приходилось гадать, что именно задумывал разработчик ПО. Чтобы программу можно было читать, в исходном коде требовались подробные комментарии, но мало кто из разработчиков утруждал себя адекватным комментированием программ — часто на это просто не хватало времени. В современных языках программа фактически самодокументируема. "Избыточные" описания уменьшают необходимость в комментариях и облегчают сопровождение программы, так как намерения разработчика более очевидны. Это новая тенденция в отрасли, и мы рассмотрим примеры программного кода, где реализуется данный подход.

Современные языки громоздки и сложны. Конечно, ученым и менеджерам использовать их трудновато. Они предназначены для профессиональных программистов, обученных искусству разбиения системы на отдельные взаимодействующие части без излишних взаимосвязей (и ненужного обмена знаниями между разработчиками). Основной модульной единицей в прежних языках была функция. В них предусматривались способы, позволяющие указать, что одни функции логически связаны теснее, чем другие. В новых языках тоже применяется такая единица модульности, как функция, но программист располагает также способами

агрегирования функций. В Ada такой конгломерат называется *пакетом*. Пакет Ada может содержать данные, с которыми работают функции пакета, но программа будет иметь только один экземпляр этих данных. В C++ и Java сделан следующий шаг: их единица агрегирования — класс позволяет программисту комбинировать функции и данные таким образом, чтобы программа могла использовать любое число экземпляров данных — объектов.

Между тем само по себе применение современных языков программирования не дает никаких преимуществ. С помощью этих языков можно написать программу, которая будет не лучше программ на традиционных языках с большим числом ссылок между отдельными частями программы, трудным в понимании исходным кодом, нуждающимся в дополнительном документировании, из-за чего внимание программиста, занимающегося сопровождением ПО, будет рассеиваться на разные уровни вычислений. Именно здесь на сцену выходит объектно-ориентированный подход.

Объектно-ориентированный подход: что он дает и какой ценой?

Объектно-ориентированный подход производит впечатление на всех (ну, почти на всех). Практически каждый знает — этот подход лучше, чем предшествующий (даже если не знает, почему). Те же, на кого ООП впечатления не производит, все же не сомневаются в его эффективности. Их смущает необходимость затрат и организационных изменений: расходы на обучение разработчиков и пользователей, усилий на создание новых стандартов, рекомендаций и документации, задержка сроков реализации проектов из-за дополнительного обучения, трудности освоения новой технологии с неизбежными ошибками и неудачным стартом.

Да, риск значителен, но и отдача велика (по крайней мере, мы так думаем). Основной стимул к внедрению ООП — доступность и широкое использование языков, поддерживающих объекты. Без сомнения, C++ — один из наиболее значимых факторов.

Может быть, объектно-ориентированный подход — просто модно? Не придет ли со временем ему на смену что-нибудь еще? Дает ли он реальные преимущества? Имеет ли объектно-ориентированный подход какие-то недостатки, требует ли компромиссных решений?

Откровенно говоря, при написании небольших программ ООП (и применение классов C++) не дает никаких преимуществ. Преимущества объектно-ориентированного подхода перевешивают его недостатки только при реализации крупных программных проектов.

Если говорить о сложности программы, то нужно иметь в виду следующее:

- Сложность самого приложения
  (что именно оно делает для пользователей)
- Сложность реализации программы
  (решений разработчиков и программистов)

Со сложностью приложения трудно что-либо сделать — она определяется самой целью создания программы. Наивно ожидать, что в будущем приложения станут более простыми. Их сложность лишь возрастет из-за увеличения вычислительной мощности и пропускной способности оборудования, позволяющих справиться со все более серьезными задачами.

А вот второе, сложность реализации программы, можно взять под контроль. Каждый раз, когда принимается решение выделить ту или иную часть задачи в отдельный модуль программы, сложность программы увеличивается, поскольку необходима кооперация, координация и коммуникации между модулями. Особенно велик риск усложнения при неоправданном разбиении на модули, когда в разных модулях оказываются действия, которые должны быть вместе.

Так зачем же разбивать эти действия по разным модулям? Никто не делает это намеренно или из злого умысла. Нередко очень трудно определить, что следует разделять, а что нет. Чтобы понять такие ситуации, нужно научиться оценивать качество проектирования. Кстати, а что такое проектирование?

Чем занимается проектировщик?

Многие разработчики ПО думают, что проектирование состоит в ответе на вопросы: какие функции и действия выполняет программа, какие данные должны получаться на выходе и какие необходимы на входе. Другие добавляют сюда решение о том, какие алгоритмы нужно применять для выполнения поставленной задачи, как должен выглядеть пользовательский интерфейс, какую производительность и надежность следует ожидать от системы и т. д. Это не проектирование, а анализ.

Проектирование начинается после получения ответов на все эти вопросы. Обобщенно говоря, проектирование — это совокупность решений о том:

- Какие модули будет содержать программа. При проектировании ПО это решение позволяет получить набор функций, классов, файлов или других составляющих программу модулей;
- Как эти модули будут связаны друг с другом (кто и что использует). Это решение описывает, какие модули вызывают сервисы других модулей и какие данные участвуют в обмене для поддержки сервисов;
- За что должен отвечать каждый отдельный модуль, т. е. принимаются решения, какие действия должны объединяться в одном модуле, а какие разделяться по разным. Однако это слишком общее наблюдение, чтобы быть полезным практически. Нужны специальные методы проектирования, обеспечивающие конкретные уровни модульности.

Если подумать, сказанное применимо не только к проектированию ПО, но и к любой творческой деятельности, будь то написание музыкальной композиции, книги, письма другу или картины. В любом случае нужно решить, из каких компонентов будет состоять "продукт", как эти компоненты соотносятся друг с другом и какова роль каждого компонента в достижении общей цели. Чем сложнее задача, тем важнее проектирование для достижения качественного результата. Простое послание по электронной почте не требует тщательного планирования. Совсем другое дело — составление руководства пользователя по программному продукту.

Структурное проектирование в качестве элементов модульности использует функции. Проектировщик определяет, какие задачи должна выполнять программа, разбивает эти задачи на подзадачи, шаги, подшаги и т. д.— до тех пор, пока не получатся достаточно малые шаги. Затем каждый шаг реализуется как отдельная, предположительно независимая функция.

При проектировании, ориентированном на данные, выделяются модули, каждый из которых отвечает за обработку конкретного элемента входных или выходных данных. Проектировщик идентифицирует входные и выходные данные программы, разбивает сложные данные на компоненты до уровня достаточно небольших процессов, обеспечивающих получение выходных данных из исходных. Затем каждый процесс преобразования данных реализуется как отдельная независимая функция.

Существуют многочисленные вариации этих методов, применимых к приложениям разного типа: базам данных, интерактивным программам, приложениям реального времени и т. д.

Во всех языках высокого уровня имеются функции, процедуры, подпрограммы или другие подобные единицы модульности (например, в Коболе это параграфы). Соответственно поддерживаются методы проектирования таких модулей. Эти методологии полезны, но не устраняют проблемы сложности проектирования. Число взаимосвязей между модулями программы остается большим, поскольку модули

связаны друг с другом через данные. Ссылки на данные усложняют код модуля, делают его менее понятным. Проектировщикам (а потом и программистам, занимающимся сопровождением ПО) приходится принимать во внимание слишком много факторов, они делают ошибки, которые трудно найти и исправить.

Разработчики ПО применяют несколько критериев, помогающих им свести к минимуму сложность и взаимозависимость фрагментов программного кода. Традиционными критериями качества ПО являются сцепление (cohesion) и связность (coupling). Современные объектно-ориентированные критерии — сокрытие информации (ограничение доступа к ней) и инкапсуляция.

Качество проекта: сцепление

Сцеплением называют связанность включенных в один модуль шагов. Если для достижения конкретной цели функция выполняет с каждым вычислительным объектом одну задачу или несколько соотносящихся друг с другом шагов, то говорят, что функция обладает хорошим сцеплением. Когда функция выполняет с объектом несколько несвязанных задач или несколько задач с разными объектами, значит, функция обладает слабым (низким) сцеплением.

Для функции с хорошим сцеплением легко придумать имя — обычно это составные термины, содержащие один глагол (действие) и одно существительное (объект): например, `insertItem`, `findAccount` (если имя соответствует содержанию, что далеко не всегда имеет место). Для функций со слабым сцеплением используется несколько глаголов или существительных, например `findOrInsertItem`, когда функция находит элемент в наборе или вставляет его, когда элемент не найден (если, конечно, функции намеренно не присваивают имя `findItem`, чтобы скрыть промахи разработчика).

Избавиться от слабого сцепления (как от любого дефекта в проекте) можно только с помощью перепроектирования — изменения списка блоков (функций) и их обязанностей. В случае слабого сцепления это означает разбиение функции на несколько функций с сильным сцеплением.

Подобный подход очень часто срабатывает, но не стоит доводить его до абсурда. В противном случае получится очень много мелких функций и проектировщику (или специалисту по сопровождению ПО) придется запоминать очень много разной информации (имена функций, их интерфейсы и пр.).

Вот почему сцепление используется не само по себе. Это не очень строгий критерий. Его нужно дополнить другими критериями, однако сцепление следует принимать во внимание при оценке альтернативных вариантов в ходе проектирования.

Качество проекта: связность

Второй традиционный критерий, связность, описывает интерфейс между функцией (сервером — вызываемой функцией) и вызывающими ее функциями (клиентами). Клиенты передают функции-серверу значения исходных данных. Например, функция `processTransaction` (клиент) может вызывать функцию `findItem` (сервер), передавать ей идентификатор (ID) элемента и текст сообщения об ошибке (входные данные). Для получения корректных результатов (например, вывода найденного элемента или сообщения об ошибке) сервер полагается на корректность ввода.

Клиент доверяет результатам, переданным сервером. Так, функция `findItem` может передавать своему клиенту `processTransaction` флаг, указывающий, что элемент не найден, или индекс найденного элемента. Это вывод (выходные данные) сервера. Общее число элементов ввода и вывода сервера — есть мера связности. Нужно постараться минимизировать связность, уменьшив число элементов в интерфейсе функции.

Критерий *связности* более мощный и строгий, чем критерий сцепления. При принятии решений в процессе проектирования очень важно не разделять операции, которые должны быть вместе. В случае такого неоправданного разделения почти неизбежны избыточные коммуникации между модулями и дополнительная связность. Например, обработку ошибки транзакции следует выполнять в одном месте, а не разбивать между функциями `processTransaction` и `findItem`.

Избавиться от избыточной связности также можно перепроектированием. Нужно тщательно продумать, что должна делать каждая функция. Если часть операций над данными выполняется в одной функции, а часть — в другой, то проектировщику следует подумать о выделении обеих частей в одну функцию. Это уменьшит сложность проекта, не увеличивая числа функций программы. Например, перенос обработки ошибок из функции `findItem` в `processTransaction` устранит необходимость передачи в `findItem` текста с сообщением об ошибке и флага обратно в `processTransaction`.



Внимание Сцепление и связность подробнее обсуждены при рассмотрении примеров программного кода в главе 9.

На данном этапе читателям было бы полезно взглянуть на программу C++ и проанализировать ее с точки зрения сцепления, связности и разделения того, что должно быть в одном модуле.

Качество проекта: связывание вместе данных и функций

Каков вклад ООП в соблюдение этих критериев качества? Не забывайте, что повышение качества ПО вовсе не означает, что исходный код программы будет выглядеть элегантнее — эстетика отнюдь не уменьшает сложности программы. Повышение качества означает, что модули программы должны быть более независимыми, самодокументируемыми, а намерения разработчика — более понятными.

ООП основывается на связывании вместе данных и операций. В этой книге достаточно подробно поясняется синтаксис, позволяющий реализовать такое связывание, однако чтобы понять, для чего нужен подобный синтаксис и почему выгодно его использовать, важно взглянуть на синтаксис объекта.

Так в чем же выгода связывания данных и операций? Проблема функционального подхода к проектированию программы в том, что "независимые" функции связываются с другими функциями через данные. Например, одна функция может присваивать значение переменной, а другая — использовать это значение (`findItem` устанавливает значение индекса, а `processTransaction` данный индекс использует). Таким образом, создается зависимость между двумя функциями (и, вероятно, между другими функциями тоже).

Одно из решений проблемы — слияние двух функций. Данное решение должно использоваться, однако работает оно не всегда. Часто функция-сервер вызывается неоднократно и, вероятно, из разных функций-клиентов, так что устранение функции `findItem` не имеет смысла.

Кроме того, некоторые функции могут присваивать и использовать значение (индекс элемента может использоваться в функциях `deleteItem`, `updateItem` и т. д.). Если что-то некорректно, в небольшой программе нетрудно отследить все значения переменной и найти источник проблемы. В крупной программе дело осложняется, особенно для программиста, занимающегося ее сопровождением, который не полностью понимает задумки разработчика. Даже сам разработчик, возвращающийся к программе через несколько недель или месяцев, часто уже не в состоянии в ней разобраться и найти функцию, использующую конкретное значение.

Было бы неплохо обозначить как-то набор функций, обращающихся к конкретной переменной и модифицирующих ее, например, сгруппировав их в исходном коде. Это помогло бы специалисту по сопровождению ПО (и возвращающемуся к программе разработчику) понять намерения проектировщика. Многие проектировщики ПО так и делают, поскольку понимают важность такой характеристики исходного кода, как его самодокументирование.

Между тем часто такой подход реализовать нелегко. Функции нередко размещают в исходном коде так, чтобы их легче было найти, в частности по алфавиту. Даже если функции группируются по переменным, к которым они обращаются, то нет никакой гарантии, что релевантные функции будут объединены вместе. Если проектировщику или сопровождающему ПО программисту нужно быстро устранить какую-то проблему или добавить функцию, обращающуюся к переменной в каком-либо месте программы, то синтаксической ошибки может не быть. Программа пройдет компиляцию, результаты выполнения будут корректны. Однако программист, занимающийся сопровождением программы, может никогда не увидеть дополнительную функцию. В функциональном программировании нелегко сделать так, чтобы все функции, обращающиеся к конкретным данным, перечислялись в одном месте.

ООП решает эту проблему путем связывания значений данных и обращающихся к ним функций. В C++ данные и операции комбинируются в большие блоки — *классы*. Это позволяет не разбивать то, что должно быть объединено. Кроме того, не нужно помнить столь много разной информации о разных частях программы. Данные могут быть закрытыми, т. е. к их значениям будет обращаться только функция, принадлежащая к тому же классу. Таким образом, намерения разработчика (помещенные в списке всех функций, обращающихся к классам) выражаются явно, в синтаксической единице — описании класса. Программист, сопровождающий программу, будет уверен в том, что никакая другая функция не обращается к этим данным и не модифицирует их.

На рис. 1.5 показана взаимосвязь между двумя объектами — сервером и клиентом. Каждый объект содержит данные, методы и имеет границу. Все, что находится внутри границы, является закрытым (private) и для других частей программы недоступно, а все, что лежит вне границы,— общедоступно (public), т. е. допускает обращение из других частей программы. Находящиеся внутри границы данных невидимы извне. Методы (функции) могут быть частично внутри, а частично вне границы. Внешняя часть представляет интерфейс метода с клиентами, а внутренняя — реализацию кода, скрытого от клиента.

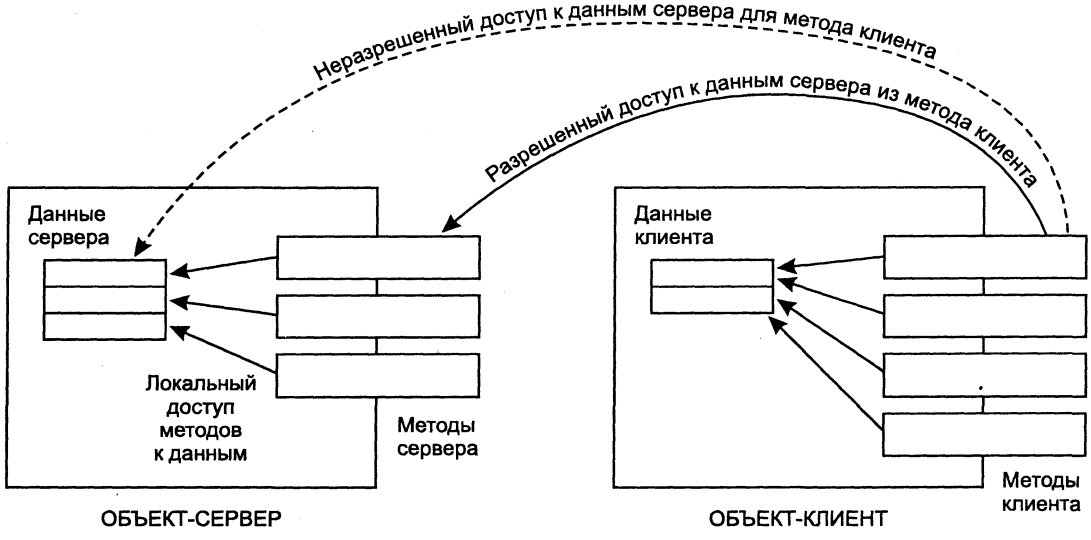


Рис. 1.5. *Взаимосвязь между объектами — сервером и клиентом*



Внимание Данная нотация была разработана Гради Бучем для проектирования и программирования на языке Ada. Она оказалась очень полезной для объектно-ориентированного программирования и проектирования. Когда методам клиента для выполнения своих операций необходимы данные сервера, они не указывают имен данных сервера. Ведь это закрытые данные, недоступные вне класса сервера. Вместо этого методы клиента вызывают метод сервера, обращающийся к данным сервера. Поскольку реализация метода сервера находится внутри сервера, обратиться к закрытым данным сервера таким образом нетрудно.

Качество проекта: сокрытие информации и инкапсуляция

При принятии решения о том, какие данные и функции должны разделяться, а какие нет, приходится выбирать из большого числа альтернативных вариантов. Некоторые из них явно плохи, другие лучше. Выбор может стать очень трудным.

Принцип инкапсуляции требует такого комбинирования данных и операций, чтобы код клиента мог выполнять задуманные действия, просто вызывая функции сервера, без явного указания имен компонентов данных сервера.

Основное преимущество подобного подхода состоит в том, что список функций, обращающихся к данным сервера, всегда доступен сопровождающему это ПО программисту в виде описания класса.

Другое преимущество заключается в том, что код клиента содержит меньше операций с данными и становится самодокументированным. Предположим, что в приложении нужно присваивать значения, описывающие заказчика,— имя, отчество, фамилию, улицу, город, штат, почтовый индекс, номер социального страхования и т. д. Всего 16 значений. Если при написании программного кода используется традиционный подход, то в коде будет содержаться 16 операций присваивания. Специалисту по сопровождению программы придется решить:

- Всем ли компонентам в описании заказчика присваиваются значения
- Присваиваются ли значения именно здесь и кому присваиваются (только компонентам описания заказчика или другим данным)

Ответы на эти вопросы могут потребовать времени и усилий. Кроме того, увеличивается сложность программы.

Если программный код написан с помощью ООП, то данные будут закрытыми, а в коде клиента нет необходимости упоминать имена описывающих заказчика переменных (имя, адрес и пр.). Вместо этого клиент просто вызывает функцию доступа, например `setCustomerData`. Намерения разработчика сразу становятся очевидными для программиста, сопровождающего ПО.

Принцип сокрытия информации требует комбинирования данных и операций, такого распределения операций, чтобы код клиента не зависел от проектирования данных.

Допустим, надо проверить не наличие переменных с названием штата и почтовым индексом, а только согласованность данных. Удобно озадачить этой работой сервер, а не клиента. Если значение почтового индекса или штата изменяется либо город становится отдельным штатом, то потребуются лишь изменения на уровне сервера. При выполнении таких операций на уровне клиента понадобилось бы менять каждый введенный почтовый индекс.

Важное преимущество ООП по сравнению с традиционным подходом — уменьшение масштабов изменений. Предположим, нужно перейти с пятизначного почтового индекса на девятизначный. При традиционном подходе пришлось бы проверять код всех клиентов, так как данные заказчика могут обрабатываться где угодно. Если программист забудет в каком-то месте изменить код для обработки большего числа знаков индекса, то это не будет синтаксической ошибкой. Проблема обнаружится лишь при тестировании.

При ООП нужно изменить функции, обрабатывающие почтовый индекс. На вызывающий эти функции клиентский код подобные изменения не влияют. Таким образом, инкапсуляция и сокрытие информации дают следующие основные преимущества:

- Благодаря связыванию функций и данных в описании класса понятно, какие функции обращаются к конкретным данным.
- Смысл клиентского кода становится более ясным из имен функций, а не из интерпретации большого числа вычислений и присваиваний нижнего уровня.
- При изменении представления данных меняется функция доступа к классу, но масштабы изменений в клиентском коде будут меньше.

Проблемы проектирования: конфликты имен

Менее критичными, но очень важными являются конфликты имен в большой программе. В программе C++ имена функций должны быть уникальными. Если один разработчик выбирает имя функции `findItem`, то другой не может назвать так какую-либо еще функцию программы. На первый взгляд проблема проста. Любой, кому потребуется такое имя, может слегка модифицировать его, например: `findInventoryItem` или `findAccountingItem`.

Многих разработчиков очень раздражает такое ограничение. Но это не просто необходимость придумывать новые имена. Это проблема дополнительных согласований между разработчиками.

Допустим, вы руководите коллективом разработчиков из 20 человек и хотите назвать функцию `findItem`. Пусть три разработчика пишут код клиента, вызывающий эту функцию. При объектно-ориентированном подходе только эти три человека и будут знать, что вы пишете функцию `findItem`. Другим такая информация не нужна — они могут сосредоточиться на других вещах.

При традиционном подходе о вашем решении придется уведомлять всех сотрудников. Конечно, они не будут посвящать свое рабочее время только изучению имен функций, придуманных вами. Тем не менее нужно будет вести (или держать под рукой) список всех функций, создаваемых разработчиками. Чтобы избежать конфликтов имен, многие организации разрабатывают сложные стандарты имен функций и тратят значительные усилия на обучение разработчиков применению этих стандартов, обеспечению согласованности. Немало времени и усилий уходит на управление изменениями при модификации стандартов.

Это может быстро превратиться в рутинную работу, отвлекающую разработчика от других вопросов. Возможности нашего внимания ограничены, и чем больше приходится запоминать, тем выше вероятность ошибки. ООП ослабляет эту проблему. Он позволяет использовать одно и то же имя функции многократно — функции должны лишь принадлежать к разным классам. Эти имена достаточно знать только разработчикам, которые данные функции вызывают. Все прочие могут направить усилия на решение других вопросов.

Проблемы проектирования: инициализация объектов

Еще одним менее критичным, но не менее важным вопросом является инициализация объектов. При традиционном подходе участвующие в вычислениях объекты программы инициализируются явно, в коде клиента. Например, клиент должен явным образом инициализировать компоненты описания заказчика. При ООП все это делается с помощью вызова `setCustomerData`. Тем не менее в коде клиента должна вызываться эта функция. Если такой вызов отсутствует, то синтаксической ошибки не будет, но будет ошибка семантическая, обнаруживаемая при

выполнении и тестировании программы. Если обработка данных о заказчике требует системных ресурсов (файлов, динамической памяти и пр.), то эти ресурсы также должны явно возвращаться в коде клиента, например путем вызова соответствующей функции.

Объектно-ориентированный подход позволяет выполнять такие операции неявно. Когда клиент создает объект, передавая необходимые данные инициализации (о синтаксисе будет рассказано далее), то нет никакой необходимости явно вызывать инициализирующие функции. При завершении вычислений выделенные объекту ресурсы возвращаются (освобождаются) без явных действий в программном коде клиента.



Внимание Это достаточно общее рассуждение. Позднее читатели познакомятся со средствами C++, позволяющими использовать эти особенности ООП. При чтении данной книги лучше иногда возвращаться к первой главе, чтобы не потерять леса за деревьями — видеть всю красоту объектно-ориентированного программирования.

Так что же такое объект?

В объектно-ориентированном программировании программа строится как набор взаимодействующих объектов, а не функций. *Объект* — это комбинация данных и некоего поведения. Программистам, конечно, известны различные термины для данных — поля данных, элементы данных или атрибуты. Говоря о поведении объекта, мы используем такие термины, как функции, функции-члены, методы или операции.

Данные характеризуют состояние объекта. Если аналогичные объекты описаны в терминах одних и тех же данных и операций, то можно обобщить идею объекта — появляется понятие класса. Класс — это не объект, а описание общих свойств (данных и операций), принадлежащих классу объектов. При выполнении программы класс не находится в памяти компьютера, как объект. Каждый объект конкретного класса содержит все определенные для этого класса поля данных. Например, каждый элемент `InventoryItem` может иметь идентификационный номер, описание элемента, имеющееся в наличии количество, закупочную цену, розничную цену и пр. Опишем эти общие свойства в определении класса `InventoryItem`. При выполнении программы создаются объекты класса `InventoryItem`, и для их полей данных распределяется память. Такие объекты можно изменять независимо друг от друга. Если изменяются значения полей данных, то говорят об изменении состояния объекта.

Все объекты одного класса характеризуются одинаковым поведением. Операции объекта (функции, методы) описываются в определении класса совместно с данными. Каждый объект конкретного класса может выполнять один и тот же набор операций. Эти операции выполняются "от лица" других объектов программы. Обычно они представляют собой операции над данными объекта: считывание значений полей, присваивание полям данных новых значений, печать значений и т. д. Например, объект `InventoryItem` может содержать среди прочего такие функции, как присваивание розничной цены или сравнение числовых идентификаторов элементов с заданным числом.

Часто программа содержит несколько объектов одного вида. Поскольку объект является экземпляром класса, термином "объект" называется каждый такой экземпляр. Некоторые называют объектом группу объектов одного вида, однако чаще для описания такой группы (потенциальных экземпляров объектов) используется термин "класс". Каждый объект одного класса имеет собственный набор полей данных, но соответствующие поля в разных объектах должны иметь одинаковые имена. Например, два объекта `InventoryItem` могут иметь одинаковые (или разные) значения розничной цены, а идентификационный номер в разных

объектах, вероятно, будет различаться. Все объекты одного класса могут выполнять одинаковые операции, т. е. отвечают на одни и те же вызовы функций. Когда вызывается функция объекта, изменяющая состояние этого объекта или получающая информацию о состоянии объекта, говорят, что объекту передается сообщение.

Это очень важная деталь. В программе на C++ вызов функции обычно связан с использованием двух объектов. Один объект посылает сообщение (вызывает функцию), а другой — получает его (иногда говорят о нем как об адресате сообщения). Назовем объект, отправляющий сообщение, *объектом-клиентом*, а получателя сообщения — *объектом-сервером*. Подобная терминология напоминает системную архитектуру клиент/сервер, но смысл ее совсем иной. Эти термины стали применяться в объектно-ориентированном программировании намного раньше, чем приобрели популярность первые клиент-серверные системы. В данной книге много рассказывается о клиент-серверных связях между объектами в программе C++.

Как будет показано позднее, объекты в программе C++ синтаксически весьма напоминают обычные переменные — целочисленные, символьные, с плавающей точкой. Распределение памяти для них очень похоже на распределение памяти для обычных переменных: она выделяется в стеке или в динамической области (см. ниже). Класс в C++ — это синтаксическое расширение того, что в других языках называется структурами или записями, позволяющими сгруппировать компоненты данных. Класс C++ включает в себя описания данных и функций.

Таким образом, когда клиенту нужно использовать объект, в частности сравнить идентификационный номер (ID) с заданным значением или присвоить элементу значение розничной цены, то имен полей данных указывать не нужно. Вместо этого вызывается предусмотренная объектом функция. Такие функции выполняют для клиента всю работу: сравнивают ID или присваивают значение цены.

Преимущества применения объектов

На первый взгляд, указание имен функций вместо имен полей данных объекта дает небольшой выигрыш, на самом деле это не так. Опыт показывает, что проектирование структур данных — более подвержено изменениям, чем проектирование операций. Благодаря использованию имен функций, а не полей данных, код клиента изолируется от потенциальных изменений в объекте-сервере. Тем самым повышается сопровождаемость программы, а это одна из важных целей ООП.

Кроме того, когда клиент вызывает функцию сервера, например `compareID`, намерения проектировщика сразу ясны программисту, занимающемуся сопровождением ПО. Если код клиента получает идентификаторы у объектов и выполняет с ними операции, это означает, что смысл кода придется уяснять из элементарных операций, а не имен функций.

Короче говоря, цель ООП та же, что и других методов разработки ПО — повышение качества ПО в соответствии с потребностями конечного пользователя (функциональности программы, общей стоимости ее разработки и сопровождения).

Сторонники объектно-ориентированной технологии надеются, что она позволит сделать программный код менее сложным. Это, в свою очередь, способно уменьшить число ошибок в ПО, повысить продуктивность его разработки и сопровождения.

Сложность ПО можно уменьшить путем разбиения программ на относительно независимые части с незначительным количеством ссылок на другие фрагменты. В таких частях нетрудно разобраться по отдельности. Применяя в качестве программных единиц классы, также можно сократить число взаимосвязей между частями программы, однако придется увеличить число взаимосвязей между частями класса: функции-члены класса работают с одними и теми же данными.

Но и это не проблема: разработкой класса обычно занимается один человек, а потому не потребуются многочисленные согласования со всеми возможными упущениями и непониманием. Ослабление зависимости между классами сводит к минимуму потребность в координации между участниками проектной команды, отвечающими за разные классы, и уменьшает число ошибок.

В отличие от взаимосвязанных частей программы независимые классы можно легко использовать повторно в другом контексте. Это увеличивает продуктивность разработки системы, а, возможно, и продуктивность реализации нескольких проектов.

Взаимодействующие части ПО приходится изучать в совокупности, а это процесс длительный и способствующий возникновению ошибок. Независимые классы гораздо проще в понимании, что повышает продуктивность при сопровождении программного обеспечения.

Конечно, ООП не освобождает от риска и расходов. Разработчиков, пользователей и менеджеров придется обучать, а стоимость такой подготовки значительна. Кроме того, сроки реализации объектно-ориентированных систем обычно больше, особенно на первых этапах проекта (анализ и проектирование). Объектно-ориентированные программы содержат больше строк исходного кода, чем обычные. (Пугаться не стоит — речь идет именно о строках исходного, а не объектного кода. Размер выполняемого кода, как правило, не зависит от методологии разработки.) Что еще более важно, языки, поддерживающие объектно-ориентированное программирование, достаточно сложны (особенно C++). Их применение связано с определенным риском: преимущества не всегда достигаются. Объектно-ориентированные программы могут быть сложнее, медленнее обычных, вызывать трудности при сопровождении. В данной книге рассказывается, как всего этого избежать и правильно использовать C++. Должное применение такого мощного и стимулирующего языка, как C++ поможет реализовать все обещания ООП.

Характеристики языка программирования C++

C++ — надстройка над языком программирования C. Сам язык C — это наследие нескольких поколений более ранних языков. Он создавался для достижения противоречивых целей. Вот почему C++ содержит несогласованные и иногда раздражающие средства. В данном разделе вкратце рассказано об основных характеристиках C, поясняется, как создатели C++ использовали это наследие для достижения своих целей.

Цели языка C: производительность, удобочитаемость, красота и переносимость

Первая цель создания языка C состояла в том, чтобы предоставить язык программирования разработчикам высокопроизводительных систем. Вот почему в C и C++ не поддерживается проверка ошибок этапа выполнения. Это способно привести к некорректному поведению программы, но может выявляться программистом в процессе тестирования. В C и C++ имеются операторы низкого уровня, эмулирующие команды ассемблера и позволяющие программисту непосредственно управлять ресурсами компьютера — регистрами, портами, масками, флагами и пр.



Внимание Те, кто не знает, что означают регистры, порты и флаги, могут не беспокоиться. Это не помешает осваивать C++. Просто судьба была к вам благосклонна и избавила от такой работы, как сотни часов мучительной отладки программ на ассемблере.

Вторая цель создателей языка C состояла в том, чтобы предложить разработчикам ПО язык высокого уровня для реализации сложных алгоритмов и структур данных, поэтому языки C и C++ позволяют программистам использовать циклы, условные операторы, функции и процедуры. Именно поэтому в C и C++ поддерживается обработка различных типов данных, массивов, структур и динамического управления памятью. (Если вам эти термины не знакомы, не беспокойтесь — изучению C++ по данному руководству это не помешает.) Эти средства обеспечивают удобство обслуживания и чтения программного кода.

Третья цель создания языка C — позволить разработчикам ПО писать элегантный и эстетически приятный исходный код. Не вполне понятно, что, собственно, означает — эстетическая элегантность. Вероятно, каждый понимает ее по-своему, но можно сказать, что программа должна быть сжатой, компактной и выполнять много операций в нескольких строках хорошо продуманного кода. Как следствие, язык предоставляет программисту значительную свободу написания программного кода, не считая его синтаксически некорректным. К этому вопросу мы вернемся позднее.

Четвертая цель языка C — поддержка переносимости программы на уровне исходного кода. Не следует ожидать, что выполняемая программа на C или C++ будет работать в разных ОС и на разных аппаратных платформах. Теоретически один и тот же исходный код должен поддерживаться несколькими компиляторами и разными платформами без каких-либо изменений. Точно так же, предполагается, что для получения программы, работающей на разных платформах, изменений не потребуется.

Первых трех целей удалось достичь, несмотря на их противоречивость. Язык C использовался для разработки операционной системы UNIX, которая со временем стала очень популярной и была реализована на большом числе аппаратных платформ, включая однопользовательские (ПК) и многопользовательские среды (мэйнфреймы, мини-компьютеры и ПК-серверы). Язык C применялся для реализации системных утилит, баз данных, текстовых редакторов, электронных таблиц и многочисленных приложений.

Конфликт между удобством чтения и компактностью программы (эстетика программирования) разрешен не был. Разработчики, ценившие удобство чтения программы, учились использованию C для получения читабельного исходного кода. Те, кто ценил выразительность, пытались получать более компактный код и даже устраивали соревнования по написанию особо выразительного (но не вполне понятного) кода.

Четвертая цель, переносимость программ C, также была достигнута, но со значительными оговорками. Сам по себе язык переносимый: если его операторы скомпилировать в разных ОС, то программа будет выполняться одинаково. Весь фокус в том, что программа может содержать не только операторы языка C, но и многочисленные вызовы библиотечных функций.

Неявная цель языка C состояла в создании компактного языка. Первоначально в нем было всего 30 ключевых слов. В результате получился язык с маленьким "словарным запасом". В нем не было операций возведения в степень, операций копирования строк текста и даже ввода и вывода. Все это (и многое другое) можно было сделать с помощью библиотечных функций, поставляемых с компилятором. Разработчики языка предоставили поставщикам компиляторов свободу применения библиотечных функций для сравнения или копирования текста, выполнения операций ввода-вывода и т. д.

Идея неплохая. Вся проблема в переносимости. Если разные компиляторы и платформы используют различные библиотечные функции, то программу не перенести на другую платформу, не изменяя вызовов функций. Ее нельзя даже перекомпилировать с помощью другого компилятора. Кроме того, такой подход затрудняет "переносимость" не только программы, но и программиста. Если он изучил одну библиотеку, придется осваивать другую.

Проблема весьма существенная, поэтому поставщики компиляторов для разных платформ признавали важность разработки "стандартной" библиотеки, которую программисты могли бы использовать на разных машинах без значительных модификаций программного кода. Предполагается, что определенные изменения все же потребуются. Из-за отсутствия единого центра стандартизации было разработано несколько версий ОС UNIX, компиляторов и библиотек для машин разных производителей. В разных ОС и на разных платформах эти компиляторы и библиотеки ведут себя по-разному.

Национальный институт стандартов США (American National Standards Institute, ANSI) провел определенную работу по стандартизации языка С в 1983–1989 гг. с целью обеспечения его переносимости. В ANSI-версии языка С были реализованы некоторые новые идеи, но с соблюдением требований обратной совместимости, так что унаследованный код С можно было перекомпилировать с помощью нового компилятора.

Сегодня, хотя исходный код С в основном переносим, проблемы остаются: перенос программы на другую машину или ОС может потребовать изменений. Навыки программистов, знающих язык С, также в основном "переносимы": можно перейти к другой среде разработки, почти не переучиваясь (но некоторая переподготовка все же потребуется).

Разработчики языка С не увидели проблемы появления разных библиотек. Платить за эту "гибкость" приходится дополнительными расходами на перенос программного кода и переобучение программистов. Опыт, накопленный отраслью при разрешении этих проблем,— одна из причин столь пристального внимания разработчиков языка Java унификации стандартов. Язык Java очень строг и помечает многие идиомы С как синтаксические ошибки. При разработке Java вопрос обратной совместимости с С был одним из низкоприоритетных. Очевидно, разработчики Java не захотели продолжать эту линию.

Цели языка C++: классы и обратная совместимость с языком С

Одной из целей создания C++ было расширение возможностей языка С за счет объектно-ориентированного подхода к программированию. "Расширение" здесь не стоит понимать буквально. C++ проектировался для 100-процентной обратной совместимости с С: все программы на языке С являются вполне "законными" программами C++. (Некоторые исключения есть, но они не столь важны.) Следовательно, C++ наследует из С все особенности и средства, хорошие или плохие ("...пока смерть не разлучит нас").

Аналогично С, язык C++ — лексемно-ориентированный и различает регистр символов. Компилятор разбивает исходный код на слова-компоненты независимо от позиции в строке — их не надо расставлять по определенным столбцам, как в Фортране или Коболе. Он игнорирует все пробелы между лексемами, и программисты могут использовать это, чтобы сделать исходный код более читабельным. Различие регистра символов помогает избежать конфликтов имен, но приведет к ошибкам, если программист не обратит внимания на разницу в написании имен (или у него просто не хватит на это времени).

В C++, как и в С, имеется всего несколько базовых типов числовых данных — меньше, чем в других современных языках. Кроме того, некоторые из этих базовых типов имеют разные диапазоны на разных машинах. Еще более осложняет дело то, что программисты могут использовать так называемые модификаторы типов, изменяющие допустимый диапазон значений на конкретной машине. Это влияет как на переносимость ПО, так и на удобство его сопровождения.

Чтобы компенсировать скудость типов данных, C++ поддерживает их агрегирование — создание сложных типов: структур, массивов, объединений, перечислений. Совокупности данных можно комбинировать в другие совокупности. Это средство также заимствовано из языка С.

C++ поддерживает стандартный набор конструкций управления ходом выполнения программы: последовательное выполнение операторов и вызовов функций, итеративное выполнение операторов и блоков операторов (`for`, `while`, циклы `do`), операторы перехода по условию (`if`, `switch`), безусловные переходы (`break`, `continue` и, конечно, `goto`). Это такой же набор, как в C, но есть некоторые различия в использовании циклов `for`.

C++, как и его предшественник,— язык с блочной структурой. Неименованные блоки кода допускают вложенность любого уровня, а переменные вложенных блоков "невидимы" для внешних. Это позволяет программистам, пишущим вложенные блоки, использовать любые имена локальных переменных, не опасаясь конфликтов имен (а, стало быть, нет необходимости координировать названия переменных с программистами, пишущими другие блоки).

Но функции C (и C++), например именованный блок, не допускают вложенности в другие функции, а имена функций должны быть в программе уникальны. Это серьезное ограничение. Оно требует координации между программистами на этапе разработки и усложняет сопровождение программы. В C++ данная проблема частично исправлена за счет введения области действия класса. Методы класса (т. е. определенные в классе функции) должны быть уникальны только в классе, а не в программе.

Как и функции в языке C, функции C++ могут вызываться рекурсивно. Другие языки рекурсивный вызов не поддерживают, поскольку рекурсивные алгоритмы используются достаточно редко. Непродуманное применение рекурсии может привести к излишним затратам времени и памяти при выполнении программы. Между тем есть несколько алгоритмов, где рекурсия особенно полезна, а потому в современных языках программирования (но не в языках сценариев — скриптах) — это стандартное средство.

Функции C++, как и функции C, могут помещаться в один или несколько исходных файлов, компилируемых и отлаживаемых отдельно, что позволяет разным программистам независимо работать над разными частями проекта. Скомпилированные объектные файлы можно позднее скомпоновать, получив выполняемый объектный файл. Очень важное свойство для разделения работ при реализации крупных проектов.

C++ — язык со строгим контролем типов, что очень напоминает C. Использование значения одного типа там, где предполагается значение другого типа (например, в выражениях или при передаче аргументов функции), дает ошибку. В настоящее время это общий принцип языков программирования. Многие ошибки типов данных можно выловить уже на этапе компиляции, а не при выполнении, что экономит время при тестировании и отладке.

В то же время C++ является языком со слабым контролем типов (да, не удивляйтесь — и сильный и слабый одновременно) и даже в большей степени, чем C. В выражениях и при передаче параметров осуществляется преобразование типов — без каких-либо сообщений. Это существенный отход от других современных языков программирования, который может привести к ошибкам, не распознаваемым на этапе компиляции. Кроме того, C++ поддерживает преобразование между родственными классами. С другой стороны, такая особенность дает возможность использовать превосходный метод программирования — полиморфизм, а компилятор не будет считать за ошибки намеренную подстановку типов.

C++ унаследовал от C применение указателей. Здесь они используются в трех целях: для передачи параметров из вызывающей функции в вызываемую, для динамического распределения памяти из динамической области (heap), если необходимо организовать, например динамические структуры данных, и для работы с массивами в компонентах-массивах. Все методы работы с указателями могут вести к ошибкам. Такие ошибки особенно трудно обнаружить, локализовать и исправить.

Как и С, язык С++ проектировался из соображений эффективности. Границы массивов не проверяются ни на этапе компиляции, ни на стадии выполнения. Программист сам должен позаботиться о целостности программы и избежать порчи других областей памяти из-за неверного индекса элемента массива — весьма распространенный источник ошибок в программах на С/С++.

Подобно языку С, С++ разработан для написания компактного исходного кода. Здесь придается особый смысл таким знакам пунктуации, как звездочки, знаки равенства, скобки, запятые и т. д. Эти символы могут иметь в программе на С++ разный смысл. Их значение зависит от контекста, что затрудняет изучение С++ по сравнению с другими языками.

В С++ добавлен ряд новых средств. Наиболее важное — поддержка объектов. В С++ структуры С расширены до классов, позволяющих связать вместе в одной единице кода функции и данные. Применение классов способствует сокрытию информации за счет локализации представления данных в определенных границах. Вне класса компоненты данных будут недоступны. Классы поддерживают инкапсуляцию. Для этого предлагаются функции доступа (методы), вызываемые клиентом. Применение области действия класса уменьшает вероятность конфликтов имен в программе С++.

Классы способствуют также иерархическому подходу к проектированию — классы более высокого уровня повторно используют классы низкого уровня. Композиция и наследование классов позволяют программисту строить сложные модели реального мира и легко манипулировать компонентами программы.

В языке есть и ряд других средств, помогающих проектировщику выразить свои намерения в самом коде программы, а не в комментариях.

Между тем, как и С, язык С++ разработан для опытных программистов. Компилятор не пытается думать за программиста, предполагая, что разработчик программы знает, что делает. Если разработчик неаккуратен, программа на С++ может получиться весьма сложной и трудной для чтения, модификации, сопровождения. Преобразование типов, операции с указателями, обработка массивов и передача параметров часто становятся источниками трудно обнаруживаемых ошибок.

К счастью, приемы программирования, о которых рассказано в данной книге, помогают понять, что нужно делать и как избежать ненужной сложности.

Итоги

В данной главе рассмотрены различные варианты выхода из "кризиса ПО". Применение объектно-ориентированных языков кажется наиболее эффективным способом предотвращения срыва сроков проекта, выпуска урезанных версий или перерасхода бюджета. В то же время писать программу на объектно-ориентированном языке труднее, чем на традиционных процедурных языках. При корректном использовании объектно-ориентированные языки упрощают чтение программы, но не ее написание. И это очень хорошо. Ведь исходный код пишется только один раз, а разбираться в нем приходится неоднократно — при отладке, тестировании и сопровождении программы.

Объектно-ориентированный язык С++ позволяет объединить данные и функции в новой синтаксической единице — классе, расширяющем концепцию типа. Благодаря классам, программу на С++ можно писать как совокупность взаимодействующих объектов, а не как набор функций. Применение классов способствует модульности, созданию фрагментов кода с сильным сцеплением и слабой связностью. Классы поддерживают инкапсуляцию, композицию классов и наследование. Это способствует повторному использованию программного кода и удобству его сопровождения. Применение классов устраняет конфликты имен и облегчает понимание исходного кода.

Важно научиться использовать C++ корректно. Неразборчивое применение в программе на C++ средств, унаследованных из C, может легко свести на нет преимущества ООП. Обсуждение этих средств до сих пор было достаточно беглым и ознакомительным. Позднее будет показано, как лучше всего их использовать — со всеми техническими деталями. Как уже говорилось выше, прежде чем вдаваться в детали синтаксиса, полезно время от времени возвращаться к данной главе, убеждаясь, что мощные объектно-ориентированные принципы языка еще не упущены из виду.

Глава 2

Быстрый старт: краткий обзор C++

Темы данной главы

- ✔ Базовая структура программы
- ✔ Директивы препроцессора
- ✔ Комментарии
- ✔ Описания и определения
- ✔ Операторы и выражения
- ✔ Функции и вызовы
- ✔ Классы
- ✔ Применение средств разработки программного обеспечения
- ✔ Итоги

В данной главе вкратце рассмотрены основные конструкции программирования на языке C++. В последующих главах они обсуждаются подробнее. Поскольку C++ — достаточно большой язык, то "краткий" обзор означает не рассказ собственно о языке, а описание действительно важных его средств, хотя оно может оказаться и не таким уж кратким. Тем не менее попробуем найти разумный компромисс.

Изучение такого языка, как C++, по предлагаемым им средствам не даст общей картины, позволяющей связать различные средства в единое целое. Многие из них нельзя обсуждать по отдельности. Именно поэтому необходима вводная глава, которая познакомит читателей с самыми важными концепциями и конструкциями языка C++, даст возможность написать первые программы на C++ и подготовиться к более полному изучению этих принципов и методов.

В программах, приведенных в книге, применяется стандарт C++ ISO/ANSI. В эту версию языка добавлены новые средства и изменен синтаксис некоторых существующих средств. Между тем во многих компиляторах реализована лишь часть средств нового языка. Вряд ли можно обсудить различия в реализации стандарта C++ подробно — слишком много разных поставщиков и разных версий компиляторов. В конечном счете старые компиляторы будут заменены новыми версиями, однако в течение многих лет отрасли придется иметь дело с программами на C++, соответствующими старым стандартам. Поскольку одной из важных целей C++ является обратная совместимость, такой старый код будет поддерживаться

новыми компиляторами. Добавление в стандарт C++ новых средств не отменяет старых. Как правило, здесь описывается новый синтаксис C++, хотя явно об этом не упоминается. Где необходимо, дается краткая ссылка на старые способы написания программного кода, что позволит читателям увереннее чувствовать себя с "унаследованными" программами.

Базовая структура программы

Листинг 2.1 показывает исходный код вашей первой программы на C++. Эта программа знакомит читателей с миром C++ (как любая первая программа в книге по программированию). Кроме того, она демонстрирует больше средств, чем обычная классическая программа, показывающая сообщение "Hello, World". Здесь выполняются некоторые вычисления, а на печать выводится значение pi (3,1415926) в степени 2.

Листинг 2.1. Первая программа на C++

```
#include <iostream>                                    // директива препроцессора
#include <cmath>                                       // директива препроцессора
using namespace std;
// директива компилятора
const double PI = 3.1415926;                           // определение константы

int main(void)                                         // функция возвращает целое значение
{
   double x=PI, y=1, z;                                // определение переменных
   cout << "Добро пожаловать в мир C++!" << endl;      // вызов функции
   z = y + 1;                                          // операция присваивания
   y = pow(x, z);                                      // вызов функции
   cout << "В этом мире pi в квадрате равно " << y << endl;
   cout << "Приятного дня!" << endl;
   return 0;
}                                                      // конец блока функции
```

Пусть вас не смущает, если программа кажется не совсем понятной. К концу данной главы вы будете понимать все детали не только этой программы.

Аналогично другим современным языкам программирования, C++ позволяет давать компилятору команду в виде удобочитаемого исходного кода. Компилятор C++ транслирует исходный код в машинно-читаемый объектный код. При исполнении программы последовательно выполняются команды машинного языка, и получается результат.

Большая часть вычислений производится со значениями, хранимыми в памяти компьютера. Мы будем рассматривать память компьютера как массив ячеек, содержащих значения. К ячейкам нельзя обращаться по хранящимся в них значениям. Это делается с помощью числовых адресов (в объектном коде) или по символическим именам (в исходном коде). Например, первый оператор программы C++ содержит:

```
z = y + 1;
```

Он указывает компилятору, что нужно извлечь значение, хранящееся в ячейке y, увеличить его на 1 (без изменения содержимого ячейки) и поместить результат в ячейку z. Реальные адреса ячеек определяются в выполняемом, а не в исходном коде. Программист работает с этими символическими именами — его не интересует, какие адреса памяти компилятор присваивает каждому имени.

В реальной памяти целым значениям, числам с плавающей точкой и символам (тексту) выделяется разное число битов и байтов, а во время выполнения эти биты обрабатываются по-разному. Для корректной генерации выполняемого кода компилятору нужно понимать намерения программиста. Вот почему перед выполнением оператора `z = y + 1;` компилятору необходимо сообщить, что `y` и `z` действительно представляют собой символические имена ячеек в памяти (а не функций, к примеру), и что хранимые в памяти под этими именами значения имеют тип `double` (один из типов C++ для вещественных чисел).

Таким образом, большую часть исходного кода составляют определения объектов, с которыми работает программа (здесь это имена `x`, `y` и `z`, другие задаются в директивах `#include` и `#define`), или описания того, что нужно сделать с объектами (сложение, присваивание, передача параметра функции).

Исходный код программы C++ может быть обычным текстовым файлом, созданным в текстовом редакторе типа Emacs или Vi в UNIX, Edt в VMS или в интегрированной среде разработки (Integrated Development Environment, IDE) на PC или Mac. Мы предполагаем, что он сохраняется в файле на жестком диске.

Обычно файлам с исходным кодом можно присваивать какие угодно имена, а вот использование расширения имени ограничено. В зависимости от компилятора, файлы исходного кода могут сохраняться с расширением `.cc`, `.cpp` или `.cxx`. Применение других расширений также возможно, но менее удобно. Если используются стандартные расширения, нужно задавать только имя исходного файла, а расширения средства разработки добавляют автоматически. Нестандартные расширения придется указывать явно.

В файле исходного кода могут определяться несколько функций (в приведенной выше программе C++ функция только одна — ее имя `main`). Нередко программа состоит из нескольких исходных файлов (эта — только из одного). Каждый исходный файл должен компилироваться для получения объектного файла. Большинство сред разработки требуют компоновки скомпилированной программы (объектных файлов). Лишь после этого программа может выполняться. (Подробнее о компоновке рассказано ниже.) На рис. 2.1 показан результат выполнения первой программы C++.

```
Добро пожаловать в мир C++!
В этом мире pi в квадрате равно 9.8696
Приятного дня!
Press any key to continue_
```

Рис. 2.1. *Вывод первой программы C++, полученной с помощью компилятора Microsoft*

```
C:\WINDOWS>echo off
Добро пожаловать в мир C++!
В этом мире pi в квадрате равно 9.8696
Приятного дня!
Press any key to continue_
C:\WINDOWS>
```

Рис. 2.2. *Вывод первой программы C++, запущенной в приглашении DOS*

Этот результат был получен исполнением файла, сгенерированного компилятором Microsoft Visual C++, Professional Edition version 6.0. Данный компилятор входит в состав пакета Microsoft Development Studio, объединяющего несколько средств разработки. Программа вызывалась приложением Development Studio. Последняя строка вывода ("Нажмите любую клавишу...") сгенерирована компилятором, а не программой. В противном случае окно сразу исчезло бы с экрана и пользователь не смог бы проверить результат. Старые версии компилятора Microsoft не добавляют данное сообщение, но все равно не удаляют окно с экрана — это должен сделать пользователь. Программа может исполняться так же, как автономное приложение, непосредственно в командной строке DOS. В этом случае последняя строка не появляется. На рис. 2.2 показан результат выполнения данной программы при запуске в DOS.

Численный результат может несколько различаться на разных машинах. Это зависит от заданного по умолчанию числа цифр в результате. C++ позволяет программисту явно задать формат вывода, не полагаясь на установки компилятора, но формат — вещь довольно сложная, и пока мы его рассматривать не будем. Примеры будут приведены ниже.

Наша первая программа C++ продемонстрировала следующие компоненты, представленные в программе C++:

- Директивы препроцессора
- Комментарии
- Описания и определения
- Операторы и выражения
- Функции и вызовы функций

В следующих разделах мы подробнее обсудим использование компонентов программы разного вида.

Директивы препроцессора

В большинстве языков компилятор видит то, что программист записывает в исходный файл. В C++ это не так. Компилятор — не первое инструментальное средство, обрабатывающее исходный код в ходе его превращения в исполняемую программу. Первый инструмент — препроцессор. Что это такое? Интересное нововведение C++, унаследованное от C. Цель препроцессора — уменьшить объем исходного кода, подготавливаемого программистом в процессе разработки ПО (или изучаемого при отладке и сопровождении).

Препроцессор обрабатывает исходный код и передает результаты компилятору. Большинство операторов программы препроцессором игнорируются и передаются компилятору без изменения. Препроцессор обращает внимание только на директивы препроцессора (и на относящиеся к ним операторы).

Директивы препроцессора начинаются с "#" и занимают всю строку. На одной строке нельзя разместить несколько директив. Если директива не помещается на одной строке, ее можно продолжить на следующей, но предыдущая строка должна заканчиваться специальным символом продолжения "\". "Решетка" (#) должна быть первым символом в строке. А как же свободный формат исходного кода C++? В предыдущей главе говорилось о том, что форматировать исходный код C++ можно любым удобным (программисту, но не компилятору) образом. Однако формально директивы препроцессора не являются частью языка C++ (или C), как препроцессор — не часть компилятора.

Конечно, на практике без директив препроцессора нельзя написать даже простую программу C++, но теоретически эти директивы — не часть языка! На практике препроцессоры поставляют производители компиляторов, но компиляторы и препроцессоры не связаны. Позднее требование было ослаблено: символ "#" должен быть не первым символом строки, а первым отличным от пробела символом.

В листинге 2.1 используются две директивы препроцессора `#include`. Эта директива приводит к прямой подстановке текста: препроцессор считывает весь файл, имя которого задается в директиве препроцессора, и заменяет директиву содержимым этого файла (без обработки, как есть). Такой подход можно использовать для комбинирования нескольких исходных файлов в одно целое. Чаще всего данная директива применяется для вставки заголовков функций, описывающих задействованные в исходном коде функции.

Имена этих заголовочных файлов заключаются в угловые скобки. Это говорит препроцессору, что нужно найти данный исходный файл в стандартном каталоге заголовочных файлов. Например, директива `#include` в первой программе задает два заголовочных файла. Первый нужен для использования функции `pow()`, второй — для операции `<<` и объекта `cout`. Подробнее о функциях, операциях и объектах рассказывается дальше. Это лишь один из примеров сложности C++. Чтобы рассказать даже о простой программе, нужно обратиться к помощи компонентов, которые будут понятны лишь при дальнейшем изучении.

```
#include <iostream>
#include <math>
using namespace std;
```

Последняя строка в этом сегменте кода из первой программы представляет директиву `using namespace`. Это не директива препроцессора, а новое средство языка. Оно указывает компилятору, что нужно распознавать код, поставляемый из заголовочных файлов. Старые компиляторы могут игнорировать эти три строки. В таком компиляторе директива `using namespace` использоваться не будет. Имя заголовочного файла в старом коде должно иметь расширение `.h`. Следовательно, первые три строки программы должны быть заменены следующими двумя строками:

```
#include <iostream.h>
#include <math.h>
```

Директива указывает препроцессору, что нужно найти в каталоге компилятора файлы include (функцию `pow()`, которая возводит в степень вещественное число, объект `cout`, представляющий стандартный вывод, например на экран, и операцию `<<`, отображающую значения на экране монитора).

Другие заголовочные файлы могут описывать функции, не являющиеся частью стандартной библиотеки. Обычно их пишут работающие над проектом программисты. Если имена этих файлов указываются в директивах `#include`, то они заключаются в двойные кавычки, например:

```
#include "c:\work\mydef.h"
```

Данная директива сообщает препроцессору, что нужно скопировать в исходный файл содержимое файла `mydef.h` из каталога `c:\work`. В приведенном примере используется абсолютное имя маршрута. Это удобно, если исходный файл перемещается в другой каталог, а заголовочный остается в прежнем. В таком случае данную директиву не нужно будет изменять. Между тем часто все дерево каталогов проекта перемещается в новое место. Если изменяется местоположение заголовочного файла, исходный файл на месте использования придется изменить. Чтобы избежать этого, программисты указывают в директивах `#include` относительный путь.

После обработки препроцессором сама директива `#include` из исходного файла убирается и не передается компилятору.

Директива `#include` очень важна. Без нее программа компилироваться не будет. Такие директивы не очень сложны — нужно знать лишь, что функция требует данного заголовочного файла. Справочные средства компилятора напомнят об этом.

Определение константы в листинге 2.1 инициализирует переменную с символическим именем `PI` значением 3,1415926. (Удобно использовать для символических констант буквы верхнего регистра — так легче будет отличить их от переменных, значения которых меняются в ходе выполнения программы.) Компилятор обработает следующую строку кода:

```
double x=PI, y=1, z;                    // определение переменных
```

Исполняемый код скопирует значение по адресу `PI` в ячейку по адресу `x`. Альтернативный метод введения константы `PI` состоит в использовании директивы `#define`. Директива `#define` применяется также для подстановки текста. Ее первый аргумент задает текст для подстановки, а второй — подставляемый вместо него текст. Когда препроцессор находит далее по тексту программы символ, соответствующий первому аргументу директивы, он заменяет его вторым аргументом директивы. В примере из первой программы C++:

```
#define PI 3.1415926
```

Данная директива сообщает препроцессору, что нужно заменить все вхождения PI на 3,1415926. Когда препроцессор обрабатывает строку кода:

```
double x=PI, y=1, z;                    // определение переменных
```

он передает компилятору следующую строку:

```
double x=3.1415926, y=1, z;
```

Обратите внимание, что препроцессор удаляет комментарий, так что компилятор его не видит. (Подробнее об этом — в следующем разделе.)

Директиву #define можно использовать для определения макрокоманд — последовательности вычислений, подставляемых в исходный код вместо простого символа (как в приведенном примере). Логически они используются в программном коде подобно функциям — ряд операций комбинируется под одним именем. Макрокоманды работают быстрее функций и очень популярны в C. В C++ вместо макрокоманд применяются встраиваемые функции (inline). Вот почему макрокоманды не обсуждаются здесь, хотя несколько лет назад программист, применяющий язык C, просто обязан был знать, как писать макрокоманды. Макрокоманды — это замечательно, но они являются источником трудно обнаруживаемых ошибок.

Другой важный набор директив препроцессора управляет условным вычислением. Директива #ifdef включает последующий исходный код, если символическое имя в данной директиве определено. Область действия данной директивы ограничивается директивой #endif. Например, следующий код включается в программу, если при компиляции имя CPLUSPLUS определено. В противном случае препроцессор убирает этот код — он не нужен, если программа компилируется как программа на языке C.

```
#ifdef CPLUSPLUS
  . . . то, что нужно для программы на C++
#endif
```

Обратите внимание, что имя CPLUSPLUS не обязано содержать значение — для #ifdef достаточно указать его в директиве #define. Отметим также, что символические имена записываются в верхнем регистре. Это не обязательно, но используется как общее соглашение по программированию. Другое популярное соглашение — имена в нижнем регистре, начинающиеся с двух символов подчеркивания.

```
#define __cplusplus
  #ifdef __cplusplus
  . . . то, что нужно для программы на C++
#endif
```

Еще один метод указания области действия директивы #ifdef — применение директивы #else. Следующий за директивой #else код (пока не встретится #endif) не включается в программу, а код после #ifdef — включается, или наоборот. Например:

```
#define MT
  #ifdef MT
  #define NFILE   40
  #else
  #define NFILE   20
#endif
```

Данный код аналогичен тому, который можно найти в заголовочных файлах. Если MT определено, то для числа файлов задается ограничение 40, а если убрать определение из исходного файла — 20.

Директиве `#ifdef` обратна директива `#ifndef` — она включает в файл последующий исходный код (до `#else` или `#endif`), если указанное в директиве имя не определено. Если имя определено, то последующий код пропускается. При наличии директивы `#else` следующий после нее код (до `#endif`) передается компилятору. Следующий пример также заимствован из заголовочного файла:

```
#ifdef NULL
  #define NULL 0
#endif
```

Это очень популярный метод, гарантирующий, что имя определено в программе и только один раз, даже если оно повторяется в нескольких файлах. Если такое определение встретится в другом файле снова, то оно просто игнорируется.

Условные директивы препроцессора часто используются для переносимости программ. Если приложение должно работать в нескольких средах, а код его в каждой среде примерно одинаков, за исключением локализованных сегментов, то эти сегменты можно включить в условные директивы. При переносе системы из одной среды в другую нужно лишь заменить директиву `#define` для одного имени на директиву `#define` для другого.

Все это звучит просто и эффектно, но реальность, конечно, сложнее, и в директивах препроцессора легко запутаться. Вот почему важно ограничить их применение простыми директивами `#include` для заголовочных файлов. Другие директивы будете использовать, когда лучше познакомитесь с языком.

Комментарии

В C++ имеются комментарии двух типов: блоки комментариев и комментарии в конце строки. Блоки комментариев начинаются со стандартного символа "/*" и заканчиваются символами "*/". Комментарии в конце строки начинаются с двух символов "//" и заканчиваются (как можно догадаться) концом строки, т. е. следующая строка — это строка исходного файла. Двузначные символы очень распространены в C++. Большая их часть унаследована из C. Применение таких символов для операций и в других контекстах вместо дополнительных ключевых слов — один из способов, позволивших создателям C разработать язык, в котором всего 30 ключевых слов (это убедит любого, что C — на самом деле очень компактный язык).

Знаки в двузначных символах (в C++ все, а не только комментарии) не должны разделяться пробелами (или другими символами).

Текст в комментариях эквивалентен пробелу и для компилятора синтаксически невидим. Это происходит потому, что препроцессор удаляет комментарии перед компиляцией исходного кода. Вот пример блока комментария:

```
/* Комментарии предназначены для человека, а не для компилятора.
   В комментарии может содержаться любой символ, включая табуляцию,
   перевод строки, //, /*. Допускается форматирование комментариев
   для облегчения чтения текста. */
```

Многие программисты применяют блоки комментариев в качестве предисловия к функции или при значительных изменениях в алгоритме. В этих комментариях описывается назначение, входные и выходные данные (результат вычислений), а также другие функции, вызываемые для выполнения задачи. Часто документируется история обновления исходного кода, указывается первоначальный автор и дата первой версии, авторы и даты изменений, цель каждого изменения. Точный

формат этих блоков в каждой организации разный. Конечно, надо придерживаться единообразного формата. А еще лучше выработать привычку составлять комментарии. Может ли быть что-нибудь более важное, чем комментарии? Да. Это обновление комментариев при изменении исходного кода. Нет ничего хуже при сопровождении программы, чем неверные комментарии.



Внимание В C допускались только блоки комментариев. Если программист хотел прокомментировать отдельные строки исходного кода, то для этого использовались блоки комментариев (так же, как в приведенной выше первой программе на C++). Похоже, создатели языка C считали, что программистов не затруднит заканчивать комментирование каждой строки символами "*/".

В листинге 2.2 показана первая программа C++, написанная для старого компилятора. Здесь для библиотечного заголовочного файла указывается расширение .h, вместо const применяется директива #define и используются блочные комментарии в стиле C.

Листинг 2.2. Первая программа C++ с блочными комментариями

```
#include <iostream.h>                                    /* директива препроцессора */
#include <math.h>                                        /* директива препроцессора */
#define PI 3.1415926
int main(void)                                           /* функция возвращает целое значение */
{
  double x=PI, y=1, z;                                   /* определение переменных */
  cout << "Добро пожаловать в мир C++!" << endl;         /* вызов функции */
  z = y + 1;                                             /* операция присваивания */
  y = pow(x, z);                                         /* вызов функции */
  cout << "В этом мире pi в квадрате равно " << y << endl;
  cout << "Приятного дня!" << endl;
  return 0;
}                                                        /* конец блока функции */
```

Чтобы избежать набора лишних символов, создатели C++ добавили в язык строчные комментарии, которые работают аналогично блочным: все, что находится между "//" и концом строки, невидимо для компилятора.

Между двумя типами комментариев в C++ есть два отличия. Первое очевидно: строчный комментарий не может занимать несколько строк, как блок комментариев. Именно поэтому блоки комментариев были введены в язык с самого начала. Впрочем, это различие не столь значительно, если учесть, что строчный комментарий может занимать всю строку.

```
// Комментарии предназначены для человека, а не для компилятора.
// В комментарии может содержаться любой символ, включая табуляцию,
// перевод строки, //, /*. Допускается форматирование комментариев
// для облегчения чтения текста.
```

Второе отличие более тонкое. Строчный комментарий может содержать любой символ, включая другие символы комментария. Завершающим ограничителем будет переход на новую строку (код ASCII 12). Блочный комментарий содержит любой символ, включая другие символы комментария, за исключением "*/". Другими словами, блочные комментарии не могут быть вложенными. Препроцессор пропустит вложенный открывающий символ "/*" как часть комментария, а вложенный "*/" интерпретирует как конец комментария, передав остальную его

часть и второй закрывающий символ "*/" компилятору. Компилятор даст сообщения об ошибках.

```
/* Здесь второй открывающий символ /* невидим */
   и компилятор интерпретирует вторую строку не как комментарий */
```

Это скорее проявление давнего непонимания программистов разработчиками компиляторов (и препроцессоров), а не особенностей применяемых инструментальных средств и их "интеллектуальности". C (и C++) благоволят к разработчикам инструментария. Кроме того, каждый программист знает, что вложенные блоки комментариев не допускаются. Так что в случае ошибки программист должен быстро понять, что произошло.

Так зачем же нужны вложенные комментарии? Часто желательно поэкспериментировать с альтернативными версиями кода, особенно когда нет уверенности в том, как они работают. Возьмем, к примеру, первую программу на C++. Что, если захочется посмотреть, как она работает без трех строк в середине? Простейший способ — закомментировать эти строки.

Листинг 2.3. Первая программа C++ с закомментированными строками

```
#include <iostream.h>                                    /* директива препроцессора */
#include <math.h>
#define PI 3.1415926
int main(void)                                           /* функция возвращает целое значение */
{
  double x=PI, y=1, z;                                   /* определение переменных */
  cout << "Добро пожаловать в мир C++!" << endl;         /* вызов функции */
/* начало исключаемого из программы блока
  z = y + 1;                                             /* операция присваивания */
  y = pow(x, z);                                         /* вызов функции */
  cout << "В этом мире pi в квадрате равно " << y << endl;
*/                                                       // конец исключаемого блока
  cout << "Приятного дня!" << endl;
  return 0;
}                                                        /* конец блока функции */
```

Препроцессор корректно обработает комментарий первой строки блока (`z = y + 1;`), но воспримет символ */ в ее конце как конец комментария, после чего передаст компилятору две следующих строки. Компилятору будет передан и символ "*/" в конце блока комментария, на что он даст сообщение об ошибке (разные компиляторы выведут разные сообщения об ошибках).

```
Compiling...

c:\data\ch02.cpp
c:\data\ch02.cpp(11) : warning C4138: '*/' found outside of comment
c:\data\ch02.cpp(11) : error C2059: syntax error : '/'

DEMO.EXE - 1 error(s), 1 warning(s)
```

Вот еще одна причина, по которой лучше использовать комментарии не в конце строки, а выделять для этого отдельную строку. Если бы это было сделано в данном примере, то все прошло бы корректно. Конечно, можно применять директивы условной компиляции, описанные в предыдущем разделе, но они сложнее, чем комментарии, могут приводить к конфликтам имен и предназначены, скорее, для переноса программы в разные вычислительные среды, чем для экспериментов с ее выполнением.

И еще пара слов о комментариях. В C++ строки представляются как последовательности символов в двойных кавычках. В таких строках символы комментария интерпретируются как литералы, а не как знак комментария. Другими словами, в строках заключенные в двойные кавычки комментарии не работают. Рассмотрим следующий оператор:

```
cout << "Hello /* there */ world << endl;
```

Этот оператор выведет не `Hello world`, а `Hello /* there */ world`.



Осторожно! Блоки комментариев не работают внутри строк в двойных кавычках. Для этого нужно вырезать текст, а не комментировать его.



Советуем Для удобства чтения могут (и должны) использоваться пробелы, разделяющие логически различные сегменты кода, однако злоупотреблять этим не следует, поскольку исходный код будет слишком растянут по вертикали.

Описания и определения

Когда программист проектирует логику вычислений, результаты одного шага часто используются в качестве исходных данных для другого. Следовательно, эти результаты должны храниться в памяти и при необходимости извлекаться снова. В приведенной выше первой программе C++ к значению y прибавляется единица, а результат используется в качестве второго аргумента при вызове функции `pow` (она возводит первый аргумент в степень второго аргумента). Чтобы сохранить значение в памяти для быстрого доступа и последующего использования, оно должно иметь физический адрес. Поскольку мы не хотим применять в исходном коде программы физические адреса, значению следует присвоить символическое имя, которое программист будет использовать в программе.

В листингах 2.1 и 2.2 сумма значения, хранимого в ячейке y, и 1 сохраняются в ячейке z. Программиста не волнует, как увязываются эти имена с адресами. Это проблема разработчика компилятора. Программист должен решить, какие значения следует хранить в памяти, и какие имена использовать для обозначения этих значений. Имя, присваиваемое программистом ячейке памяти, называется *идентификатором*. На самом деле программисту нужно придумать идентификаторы не только для переменных программы, но и для констант, функций, типов данных и меток (об этом далее будет рассказано более подробно).

Синтаксические правила для идентификаторов просты: они могут начинаться только с буквы или подчеркивания ("_"), но не с цифры или другого специального символа. Иными символами идентификаторами могут быть буквы A–Z, a–z, цифры 0–9 и знаки подчеркивания. Теоретически общее число символов не ограничивается. На практике компилятор может не различать идентификаторы, у которых совпадают первые 31 символ (это старое ограничение на длину идентификатора). Если 31 символа недостаточно, следует поискать более короткие имена.

За исключением подчеркивания, других специальных символов (&, $, # и т. д.) в идентификаторах не допускается. Не разрешается также использовать внутри идентификатора пробелы.

Вполне допустимо начинать имя идентификатора с подчеркивания, но лучше избегать этого, так как с подчеркивания могут начинаться системные идентификаторы. Применяйте подчеркивание только внутри идентификатора для разделения компонентов его имени (`sum_of_squares`). Еще один популярный метод состоит в том, чтобы начинать новый компонент имени с буквы в верхнем регистре (`SumOfSquares`).

Признаком хорошего вкуса и благоразумия считаются *мнемонические* имена идентификаторов. Это означает, что имя идентификатора связано с его назначением в программе. С этой точки зрения имена в нашей первой программе на C++ (x, y, z) не особенно хороши. Их можно применять, если данные значения используются в программе всего несколько раз и в относительно простых вычислениях. Имя PI лучше — оно говорит о смысле значения (по крайней мере тем, кто знает, что это такое).



Внимание В C++ различается регистр символов. Это означает, что компилятор будет различать такие идентификаторы как cnt, Cnt и CNT. Все это — разные имена, что нередко бывает источником ошибок в программе, особенно, если программисты раньше работали с языками, не различающими регистр символов.

Как уже упоминалось выше, определяемые в программе константы часто обозначаются буквами в верхнем регистре. Так лучше отличать их от объектов, значения которых при выполнении программы меняются (таких, как PI в приведенной выше программе C++).

Ключевые слова C/C++ зарезервированы и не могут использоваться в программе в качестве идентификаторов. Все они записываются буквами в нижнем регистре. Вот список ключевых слов, общих для C и C++:

```
auto break case char const continue default do double else enum extern float
for goto if int long register return short signed sizeof static struct
switch typedef union unsigned void volatile while
```

А вот ключевые слова, зарезервированные в языке C++, но не в C:

```
asm bool catch class const_cast delete dynamic_cast explicit export false
friend inline mutable namespace new operator private protected public
reinterpert_cast static_cast template this throw true try typeid typename
using virtual wchar_t
```

Не пытайтесь сейчас запомнить все эти идентификаторы. Более того, если попытаться использовать данные ключевые слова как идентификаторы переменных, компилятор сообщит, что делать этого нельзя. Так что данный список приведен не для того, чтобы запоминать эти ключевые слова, а чтобы показать, почему компилятор "жалуется", если их применять в качестве идентификаторов.

В других языках с менее строгими ограничениями программист может ввести подходящий идентификатор и указать его с левой стороны присваивания, особенно не задумываясь: `sum_of_squares = 0;`

Если сделать это в программе на языке C++, то компилятор сообщит, что `sum_of_squares = 0;` — необъявленный идентификатор. В C++ это синтаксическая ошибка. Прежде чем использовать имя для переменной в C++, его нужно определить.

Имена переменных обозначают адреса в памяти компьютера, где содержатся типизированные значения. Эти значения могут изменяться при выполнении программы.

А как с именем PI в приведенной выше программе? В листинге 2.1 оно определено как константа, и попытки изменить данное значение (например, `PI = 0;`) дадут синтаксическую ошибку. В листинге 2.2 это также константа, заданная в директиве препроцессора `#define`. После подстановки значения препроцессором оно становится неизменяемым значением-константой. Например, оператор `PI = 0;` будет преобразован в `3.1415926 = 0;`, а это синтаксическая ошибка.

Переменные в памяти содержат *типизированные* значения, т. е. программист должен определить тип значения, хранимого в переменной. Тип переменной описывает диапазон допустимых для нее значений и разрешенные операции с этими

значениями. Определение устанавливает связь между идентификатором и типом. Каждое определение заканчивается точкой с запятой, например:

```
int num;
double sum_of_squares;
char letter;
```

Первое определение говорит о том, что идентификатор num обозначает целочисленную переменную. Его размер — четыре байта (32 бита), а диапазон значений — от –2147483648 до +2147483647 (позднее поясняется, что размер типа зависит от машины).

Для данного типа действительны четыре арифметические операции, взятие по модулю (получение остатка), сравнения, сдвиг влево или вправо, логические операции, инкремент и декремент (положительное и отрицательное приращение, об этом подробнее — см. главу 3).

Во втором определении говорится, что идентификатор sum_of_squares используется для переменной двойной точности с плавающей точкой. Она имеет размер до 8 байт, а абсолютное значение достигает 1.7976931348623158e+308 (здесь e+308 означает 10 в степени 308 — довольно большое число). Для данного типа предусмотрены четыре арифметические операции, инкремент и декремент, сравнения.

В третьем определении используется ключевое слово char, означающее, что идентификатор применяется для символьной переменной и может содержать значения, представляющие символы (согласно кодам ASCII). Что касается операций, то в C++ символы интерпретируются как целые значения одинарной точности.

Целые значения могут применяться в счетчиках и математических вычислениях. Целочисленные операции — на любой машине самые быстрые. Вот почему целые используются во всех случаях, где достаточно их диапазона и точности. Числа с плавающей точкой имеют дробную часть. Они применяются в коммерческих и научных расчетах, где точности целых чисел недостаточно (или мало диапазона в два триллиона).

Принцип типа является фундаментальным и в функциональном, и в объектно-ориентированном программировании. Каждое значение, обрабатываемое в программе C++, должно иметь тип. Если тип используется некорректно, компилятор помечает это как синтаксическую ошибку. Такое может случиться, если один тип применяется там, где предполагается значение другого типа. Программист должен позаботиться о корректном использовании типов в программе C++ — это одна из его основных задач.

В C++ применяются только встроенные типы данных, т. е. типы данных, уже доступные в самом языке. Это целые (например, 65), числа с плавающей точкой (65.0) или символьные значения ("a").

Все они являются *примитивными* (или скалярными) типами, т. е. не допускается декомпозиция значения этих типов на компоненты, с которыми в программе можно манипулировать отдельно. Например, число двойной точности с плавающей точкой имеет целую и дробную части (а также экспоненциальную часть), но язык C++ не предусматривает возможности отдельного доступа к этим частям. Можно обращаться только ко всему значению. Таким образом, типы C++ — инструмент абстракции. Они позволяют сконцентрироваться на том, *что* нужно делать со значением, а не на том, *как* манипулировать с отдельными компонентами.

Бедность фундаментальных типов в C++ частично компенсируется введением некоторых вариаций целочисленных типов и типов с плавающей точкой — по размерам, диапазонам и точности. Однако это не намного меняет ситуацию. Более важно, что C++ поддерживает составные значения, агрегаты данных, позволяющие комбинировать отдельные значения примитивов в массивы, структуры и классы. Значения составных типов включают в себя несколько компонентов, и C++ предусматривает методы доступа к отдельным компонентам таких типов.

Когда определение типа обрабатывается на этапе выполнения, для переменной (примитива или составного типа) выделяется область памяти — переменная может использоваться для хранения и извлечения значений описанного для нее типа. Здесь нет ничего особенного: таким образом работают все современные языки со строгим контролем типов.

Некоторые программисты для удобства чтения используют для каждого определения отдельную строку исходного кода. Другие полагают, что это только затрудняет чтение, и помещают несколько определений на одну строку:

```
int num;    double sum_of_squares;
```

Если переменные имеют один тип, их можно определять отдельно. Каждое определение включает в себя имя типа и заканчивается точкой с запятой:

```
int a;  int b;  int c;
```

Разрешается и комбинировать эти определения: имя типа указывается лишь один раз, а имена переменных разделяются запятыми. Все определение заканчивается точкой с запятой. Например:

```
int a, b, c;                // допустимое сокращение
```

Другими словами, область действия имени типа (здесь `int`) включает в себя имена всех переменных между именем типа и следующей точкой с запятой. Переменные `a`, `b`, `c` будут целыми. Эти два стиля определения эквивалентны, но их не следует путать. Например, следующее определение является синтаксической ошибкой:

```
int a, b, int c;          // ошибка
```

А вот такие определения будут правильными:

```
int a, b;   int c;        // нет ошибки
```

Небольшая, но значительная разница. Программист, работающий с C++, всегда должен помнить о таких различиях.

Для программиста разница между запятой и точкой с запятой критическая. Не путайте их.

Многие программисты определяют переменные в начале блока функции или файла. Так сделано и в примере первой программы на C++. В языке C это единственный способ определить переменные, однако C++ позволяет программисту делать это в середине программы — ближе к тому месту, где переменные используются. Листинг 2.4 показывает, как *мог бы* выглядеть листинг 2.1, если применить более гибкий способ определения переменных.

Листинг 2.4. Первая программа C++ с определениями в середине программного кода

```
#include <iostream>                                    // директива препроцессора
#include <cmath>                                       // директива препроцессора
using namespace std;
// директива компилятора
const double PI = 3.1415926;                           // определение константы
int main(void)                                         // функция возвращает целое значение
{  cout << "Добро пожаловать в мир C++!" << endl;      // вызов функции
   double y=1, z;                                      // определение переменных
   z = y + 1;                                          // операция присваивания
   double x=PI;                                        // определение переменной
   y = pow(x, z);                                      // вызов функции
   cout << "В этом мире pi в квадрате равно " << y << endl;
   cout << "Приятного дня!" << endl;
   return 0;
}                                                      // конец блока функции
```

Для выполнения программы не важно, насколько далеко определяется переменная. Главное сделать это до ее использования. Выполнение приведенной выше программы даст тот же результат, как на рис. 2.1. Расстояние от определения до использования переменной значимо скорее для читателя, особенно если переменная используется один-два раза без какой-либо интерпретации. Если следующее использование переменной отделено от первого, а программисту, сопровождающему программу, нужно проверять определение переменной, то удобнее всего размещать определение в начале, а не в середине функции.

Еще один знакомый многим термин — это *объявление*, или *описание* (declaration). В некоторых других языках объявление и определение — синонимы. В C++ они несколько различаются — это унаследовано из языка C. Если при определении имя переменной ассоциируется с ее типом и для переменной выделяется место в памяти, то объявления лишь ассоциируют имя и тип, а память для переменной может выделяться где-либо еще. Так происходит, например, в программе, состоящей из нескольких файлов, когда переменная определяется в одном файле, а используется в другом. В этом случае с помощью ключевого слова `extern` она определяется в использующем ее файле как внешняя:

```
extern int count;
```

Теперь можно использовать переменную `count` в исходном коде данного файла. Все ссылки на переменную `count` в данном файле преобразуются в ее адрес, определенный в другом файле.

Еще одно различие определений и объявлений состоит в том, что определения должны быть в программе уникальными, а объявления могут повторяться сколько угодно раз. Например, такие определения недопустимы (даже если это желательно):

```
int a; int a;                          // синтаксическая ошибка
```

С другой стороны, допускаются следующие описания:

```
extern int count;   extern int count;       // ошибки нет
```

Хотя это выглядит не более разумно, чем предыдущий пример, но бывают ситуации, когда может потребоваться что-либо подобное, а если сделать это по ошибке, компилятор ошибку не покажет. Далее мы увидим, как такие фундаментальные принципы обобщаются для функций и типов.



Осторожно! Определение должно быть уникальным, а объявление может повторяться.

После определения (или объявления) переменных программа может работать с ними. Прежде чем использовать значения переменных в программе, им нужно присвоить эти значения, иначе возникнет такая ситуация, как применение неинициализированных переменных — весьма распространенная ошибка.

Существуют два способа снабдить переменную значением: операция присваивания и инициализация. В данном примере используются операции присваивания (каждая завершается точкой с запятой):

```
double x, y, z;
x = PI; y = 1;
```

Как можно заметить, в примере первой программы на C++ переменные `x` и `y` (но не `z`) инициализировались:

```
double x = PI, y = 1, z;
```

Результат будет таким же: переменная x получает значение 3,1415926536, переменная y — значение 1, а переменная z остается неинициализированной. Хотя мы имеем дело с переменными примитивных типов, разница между присваиванием и инициализацией с практической точки зрения не важна. Для объектов, определяемых программистом, эта разница более существенна. Когда будет обсуждаться инициализация объектов, все станет гораздо интереснее.

Переменные разрешается инициализировать только в определении, но не в объявлении. Например, переменная count может инициализироваться только в том файле, где она определена (там, где выделяется память для переменной). В файле, где переменная объявляется (как внешняя), ее можно без ограничений присваивать и обращаться к ней, но не инициализировать. Так, следующая попытка даст ошибку:

```
extern int count = 0;              // синтаксическая ошибка
```

Это было только введение в типы данных C++. В главе 3 о них рассказывается подробнее. Там же говорится об операциях со значениями разных типов.

Операторы и выражения

Оператор — программная единица, выполняемая как логическое целое, т. е. отдельные компоненты, шаги скрыты от программиста — эти детали не должны привлекать его внимание (по крайней мере, не сейчас). Оператор программы — средство абстракции. Он позволяет концентрироваться на том, *что* должно делаться, а не на том, *как* это происходит.

Описания и определения, о которых рассказывалось выше, представляют собой операторы. Детали распределения памяти нас не интересуют. Например, не важно, находится ли переменная a за переменной b или за переменной c, начинается ли слово со старшего байта и т. д. Нужно лишь знать, что память выделена для всех трех целочисленных переменных:

```
int a, b, c;
```

Показанное в предыдущем разделе присваивание представляет второй тип оператора. Целевая часть присваивания (переменная, получающая значение) находится слева, а выражение, определяющее присваиваемое переменной значение,— справа. Первая программа C++ содержала следующую операцию присваивания:

```
z = y + 1;
```

При выполнении этого оператора сохраненное в переменной y значение складывается с 1 и результат сохраняется по адресу, соответствующему переменной z. На содержимом y такая операция не отражается. Переменная изменяется только тогда, когда ее имя указывается в левой части операции присваивания.

Стоит повторить, что определения в начале листинга 2.1 содержат инициализацию переменных, но не присваивания. Хотя синтаксис похож, для объектов C++ вызываются разные функции:

```
double x=PI, y=1, z;       // x и y инициализируются, но не присваиваются
```

Правая часть операции присваивания содержит выражения. Они состоят из операций и операндов. *Операндами* могут быть переменные, числовые литералы или другие выражения. В примере первой программы C++ выражение, используемое для установки значения переменной z, содержало операнды y и 1. Если необходимо, для структурирования сложных выражений используются скобки, например:

```
z = (y + 1) * (y - 1);       // выражение с подвыражениями
```

Здесь операнды `y + 1` и `y - 1` представляют собой небольшие выражения. Каждое выражение возвращает типизированное значение, которое может использоваться в операции присваивания или в другом выражении.

В выражениях допускается применение 55 различных операций C++, включая арифметические операции "+", "-", "*", "/", сравнения, "<", ">" и др. 55 операций изучить сложно. Поскольку символов для них не хватит, в C++ часто применяются двухсимвольные операции (например, для сравнения — операция "=="). Операции разбиты на 18 уровней старшинства, что тоже нелегко запомнить. Для указания порядка выполнения операций многие программисты предпочитают использовать скобки, а не полагаться на старшинство операций. Подробнее об этом рассказано в следующей главе.

Существует важное различие между компонентами в левой и в правой части присваивания. В выражениях C++ можно указывать и имена переменных, и литеральные значения. Если задается имя переменной (`y`), то в выражении используется не ее адрес, а хранимое по этому адресу значение. Если указывается значение, то оно используется непосредственно, хотя хранится по некоторому адресу. Это кажется не очень простым, но суть такова: литеральное значение не может находиться в левой части операции присваивания. Например, следующее уравнение в C++ недопустимо:

```
1 = z - y                    // в C++ не разрешается
```

Третий тип оператора — вызов функции. В нем указывается имя выполняемой функции, используемые ею аргументы (если они есть) и возвращаемое значение (если таковое имеется).

Приведенный выше пример первой программы на C++ состоит лишь из одной функции (`main`). Она использует (вызывает) функцию `pow`. Чтобы отличать имена функций от прочих имен C++, программисты и технические писатели используют общее соглашение: независимо от числа аргументов после имени функции указываются скобки, например `main()` или `pow()`.

Библиотечная функция `pow()` использует два аргумента — возводимое в степень число и значение степени. После возведения в степень возвращаемый результат сохраняется в первом значении. Возвращаемое значение может использоваться как компонент выражения. Вот почему функция вызывается следующим образом:

```
y = pow(x,z);
```

Если первый аргумент — 3,1415926, а второй — 2, то возвращается значение 9,869604. (В других языках вычисления с нецелыми числами приближенные).

Когда функция вызывается при выполнении программы, выполнение вызывающей функции приостанавливается и начинает исполняться код вызванной функции. После завершения вызываемой функции выполнение вызывающей функции возобновляется. При вызове другой функции этот процесс повторяется.

Вызовы библиотечных функций ввода-вывода в C++ пояснить труднее, чем другие библиотечные вызовы. Они используют предопределенные классы, объекты и перегруженные (overloaded) операции, о которых подробнее будет рассказано далее. Давайте попробуем. Используем библиотечные объекты `cout` (для вывода) и `cin` (для ввода) с двумя видами двузначных операций. С библиотечным объектом `cout` применяется операция вставки "<<", а вывод направляется на экран. Для отображения используются имена переменных, литеральные числовые значения и строки в двойных кавычках. С библиотечным объектом `cin` применяется операция извлечения ">>". В этом случае ввод с клавиатуры сохраняется в переменных, имена которых заданы в качестве операндов.

Звучит сложно, но основные принципы ввода-вывода очень просты. Поскольку каждый оператор ввода-вывода должен задавать свой собственный объект (`cout` или `cin`), их нельзя использовать в одном операторе. Следовательно, операции ">>" и "<<" в одном операторе не применяются. Листинг 2.5 показывает пример ввода двух целых чисел с клавиатуры и вывода на экран их суммы.

Листинг 2.5. Интерактивная программа с операторами ввода и вывода

```
#include <iostream>
using namespace std;
int main(void)
{
  int a, b, c;                                          // определения переменных
  cout << "Введите два целых числа и нажмите Enter ";
  cin >> a >> b;                                        // два вызова функции: извлечение
  c = a + b;
  cout << "Их сумма равна " << c << endl;
  return 0;
}
```

```
Введите два целых числа и нажмите Enter: 22 33
Их сумма равна 55
```

Рис. 2.3. *Результат программы с интерактивным вводом-выводом*

Вывод этой программы представлен на рис. 2.3.

Каждое использование операций << и >> представляет вызов функции. Библиотечный компонент `endl` — это так называемый манипулятор. Каждый элемент вывода, включая строки в двойных кавычках (и символы в одинарных), литералы (числовые значения), переменные или выражения, должен иметь свой собственный оператор. То же самое относится к каждому элементу вывода. Применять запятые или пробелы для разделения компонентов ввода-вывода некорректно. Например, такой оператор даст ошибку:

```
cout << "Их сумма равна ", c endl;          // типичная ошибка: запятая и пробел
```

Еще хуже, что компилятор часто не может корректно диагностировать источник проблемы. В сообщениях об ошибках обычно говорится об отсутствующей точке с запятой, пропущенных аргументах, несоответствии параметров и других интересных проблемах. Вот еще одна причина, почему не стоит тратить особенно много усилий на расшифровку сообщения компилятора об ошибках. Их следует понимать так, что в программе имеется ошибка, и уповать на собственную логику.



Осторожно! Каждый компонент ввода и вывода должен иметь свою собственную операцию >> или <<. Применение запятых или пробелов для разделения компонентов не допускается. Не разрешается также комбинировать операции ввода и вывода в одном операторе.

Библиотека `iostream` содержит мощные и гибкие функции, однако форматированный вывод в ней достаточно разнообразен. В C++ поддерживается также альтернативный набор стандартных библиотечных функций: `printf()`, `scanf()` и их вариации. Они заимствованы из C и весьма распространены в унаследованных программах C и в коде C++. Чтобы использовать их, следует включить заголовочный файл `stdio.h`. Легче написать код для форматированного вывода с помощью этих функций, чем с помощью функций `iostream`. Между тем при применении функций `stdio.h` возникает больше ошибок. Библиотека `iostream` популярнее, чем эти старые функции — они более не являются стандартными. К сожалению, нельзя просто позабыть про библиотеку `stdio.h` и полностью переключиться на библиотеку `iostream`, так как функции `stdio.h` часто применяются в графическом интерфейсе Windows и для обработки строк.

Функции, как и операторы других типов, представляют собой инструмент абстракции. В исходном коде клиента (как, например, в приведенной выше первой программе C++) задается, *что* должно быть сделано, и без лишних описаний, *как* это делается.

До сих пор рассматривались три популярных вида операторов: определения (и объявления), присваивания и вызовы функций. Четвертым является *определение типа*. Оно комбинирует компоненты в составной тип, такой, как структура или класс. Со значениями составного типа можно работать так, как будто это единое целое. Кратко об определениях типа будет рассказано в разделе "Функции и вызовы функций". Основная же часть данной книги посвящена программированию с использованием классов.

Последний тип оператора — *составной оператор*, или *операторный блок*. Это последовательность операторов в фигурных скобках. Составной оператор может использоваться там же, где применяется обычный оператор (включая другой составной оператор). Например, в приведенном примере программы C++ можно скомбинировать три последних оператора в операторный блок (см. листинг 2.6).

Листинг 2.6. Первая программа на C++ с операторным блоком

```
#include <iostream>                          // директива препроцессора
#include <cmath>
using namespace std;
const double PI = 3.1415926;                 // определение константы
int main(void)
{
  double x=PI, y=1, z;                       // определение переменных
  cout << "Добро пожаловать в мир C++!" << endl;
  z = y + 1;
  y = pow(x, z);
  {                                           // начало операторного блока
    cout << "В этом мире pi в квадрате равно " << y << endl;
    cout << "Приятного дня!" << endl;
    return 0;
  }                                          // конец операторного блока
}                                            // конец блока функции
```

Здесь он мало что меняет: программа будет выполняться, как раньше. Однако есть немало операторов C++ (например, условия и циклы), где желательно вместо одного оператора использовать группу. Если логику вычислений нельзя сжать в один оператор, будет проблема. Применение блока операторов ее решает.

Односимвольные ограничители блока { и } должны быть парными, иначе возникнет синтаксическая ошибка. Они открывают и закрывают область действия, являясь структурным элементом программы. Функция `main()` также разграничивается фигурными скобками, как исходный код любой функции. Разница между составным оператором и телом функции в том, что составной оператор не имеет имени (это неименованный блок), а тело функции — именованный блок.

Операторы в программе C++ выполняются последовательно — сверху вниз. В идеале каждый оператор размещается на отдельной строке с таким отступом, чтобы структура программы была очевидна для читателя. Например, в листинге 2.6 операторы функции `main()` имеют отступ вправо относительно директив препроцессора и заголовка функции `main()`. Операторы во вложенном блоке сдвигаются вправо относительно других операторов функции `main()`.

Если операторы представляют собой последовательные шаги для достижения одной цели, то не возбраняется поместить их на одну строку:

```
z = y + 1; y = pow(x,z);                     // два подшага алгоритма
```

Операторы, находящиеся на одной строке, выполняются слева направо. Сколько операторов можно поместить на одну строку кода? Это зависит от удобства чтения программы — строки не должны быть слишком длинными. Однако если для

каждого оператора выделяется отдельная строка, программа вытягивается в длину. Читатель не запомнит, что он видел на предыдущих страницах — придется "листать" исходный код. Размещение нескольких операторов на одной строке делает проблему не такой острой — программа станет компактнее, хотя есть опасность пропустить действие, "спрятанное" в строке между другими операторами. По большому счету, все это дело вкуса. Во многих случаях лучше, когда операторы, размещаемые на одной строке, реализуют какую-то общую цель.

В C++ каждый оператор должен завершаться точкой с запятой. В некоторых других языках точка с запятой разделяет операторы, так что последний оператор в последовательности ее не имеет. В C++ точкой с запятой заканчивается каждый оператор. Или почти каждый — составные операторы точкой с запятой не завершаются. Если посмотреть пример программы, то видно, что после фигурных скобок точки с запятой нет.

Фактически точка с запятой может превратить любое выражение в оператор. Например, в C++ допустимо:

```
y + 1;
```

Конечно, это совершенно бесполезное выражение. Другие языки подобного не допускают. Однако в C и в C++ такая запись вполне законна. Зачем думать о том, насколько законна бессмысленная вещь, если незачем так писать? Нет, это не так уж глупо, как кажется. Если такая ошибка будет сделана случайно (например, пропущена левая часть в присваивании), компилятор ее проглотит и не сообщит о том, что что-то не так. Такая ошибка не проявится как синтаксическая (в других языках так и было бы). Она превратится в ошибку этапа выполнения и может быть выявлена только в процессе отладки.

В качестве одного из видов операторов можно рассматривать управляющие конструкции. Они изменяют ход выполнения программы. Есть три вида операторов управления выполнением программы:

- Условные операторы
- Циклы
- Вызовы функции



Внимание В данном разделе рассматривается только небольшая часть операторов управления. Подробнее о них рассказано в главе 4.

Простейший оператор условия — это оператор `if`. Он имеет общую форму:

```
if (выражение) выполняемый_оператор;
```

Синтаксически `if` представляет собой один оператор. Если `выполняемый_оператор` один, то `if` завершается точкой с запятой. В составном операторе после закрывающей фигурной скобки точки с запятой не требуется. Круглые скобки для выражения обязательны.

Следующий за выражением оператор выполняется, если выражение истинно, а если оно ложно — пропускается. Для выделения управляющих структур часто используются отступы. Ниже в сегменте кода проверяется температура по Фаренгейту. Если она выше точки замерзания, то выводится сообщение. В противном случае сообщение на экране не появляется:

```
if (fahr > 32)        // выражение обычно следует на отдельной строке
  cout << "0 запуске двигателя авто можно не беспокоиться" << endl;
```

Во второй форме оператора `if` используются две ветви: одна выполняется, если условие истинно, а другая — если ложно. Каждая ветвь содержит один

оператор (завершаемый точкой с запятой) или составной блок операторов (без точки с запятой в конце). Например:

```
if (fahr > 32)                                    // ключевого слова then в C++ нет
  cout << "О запуске двигателя авто можно не беспокоиться" << endl;
else
  cout << "Утром будьте осторожны" << endl;
```

Ключевое слово then в C++ отсутствует: оно подразумевается. Ключевое слово else в этой конструкции должно использоваться. Заметим, что отступы служат для выделения управляющей структуры.

Простейшим оператором цикла является оператор while:

```
while (выражение) выполняемый_оператор;
```

Синтаксически тело цикла — один оператор. Для простого выполняемого оператора цикл while завершается точкой с запятой. Если тело цикла — составной оператор, то точки с запятой после закрывающей правой скобки не требуется. Выражение обязательно заключается в круглые скобки.

Тело цикла, выполняемый_оператор, выполняется, если выражение истинно. Затем выражение (условие цикла) проверяется снова. Если оно становится ложным, то тело цикла пропускается и управление передается следующему оператору (когда он присутствует).

```
Числа    Их квадраты

   8       64
   9       81
   10       100
   11       121

Приятного дня
```

Рис. 2.4. *Вывод цикла, вычисляющего квадраты чисел*

Программа в листинге 2.7 вычисляет квадраты чисел 8, 9, 10 и 11, а на экран в виде таблицы выводятся сами числа и их квадраты (результат представлен на рис. 2.4). Сначала выводится заголовок таблицы и пустая строка, а затем в качестве переменной цикла применяется переменная num. Перед выполнением цикла num инициализируется значением 8 — оно используется при первом проходе цикла. В теле цикла num увеличивается на 1. В условии цикла проверяется, что num все еще меньше 12. Если это так, то выполняется тело цикла (в фигурных скобках), а значение num снова увеличивается. Выполнение цикла продолжается, пока значение num не станет равным 12. Когда условие цикла становится ложным, тело цикла пропускается и выполняется последний оператор программы.

Листинг 2.7. Пример программы с циклом и форматированием вывода

```
#include <iostream>
#include <iomanip>
using namespace std;
int main (void)
{
  int num = 8, square;                          // перед циклом инициализируется num
  cout << "Числа и их квадраты" << endl << endl;
  while (num < 12)                              // num используется как переменная цикла
  { square = num * num;                         // num используется в теле
    cout << " " << num << " " << square << endl;
    num = num + 1;                              // переменная модифицируется в конце цикла
  }
  cout << endl; << "Приятного дня" << endl;
  return 0;
}
```

Когда оператор << передает символы на экран с помощью объекта cout, они отображаются без разделяющих пробелов. При завершении преобразования

одного значения (например, num) из двоичного вида в символьный и переходе к преобразованию другого значения (например, square) промежуточные пробелы также не добавляются. Такой неформатированный вывод, конечно, неудобен. Для быстрого форматирования вывода можно вставить между компонентами промежуточные пробелы. Именно это и делает оператор cout в листинге 2.7.

Такой метод форматирования редко подходит. При разных проходах цикла выводятся значения с разным числом символов. Столбцы будут невыровненными. Проблему можно устранить с помощью манипулятора setw, задающего число позиций (по ширине) для каждого компонента. Этот манипулятор нужно вставить в поток вывода в операции << точно так же, как любой другой компонент вывода. Например, при вставке setw(4) для следующего компонента выделяется 4 позиции вывода. Если нужно форматировать несколько компонентов, то перед каждым необходим свой манипулятор setw (даже если ширина вывода для каждого компонента одна и та же).

Заменим оператор cout в листинге 2.7 на следующий:

```
cout << setw(4) << num << setw(10) << square << endl;
```

Чтобы это работало, в программу нужно включить заголовочный файл iomanip (см. листинг 2.7). Вывод данной версии программы показан на рис. 2.5. Для числовых значений вывод выравнивается вправо на заданную величину. Для строк символов он выравнивается влево. Если выводимое значение не помещается в заданных позициях, то объект cout задействует на экране столько позиций, сколько требуется, а остальная часть выводится справа. Вывод никогда не усекается из-за недостатка позиций.

```
Числа     Их квадраты

   8          64
   9          81
  10         100
  11         121

Приятного дня
```

Рис. 2.5. *Вывод цикла с заданными для каждого элемента позициями*

Нужно хорошо разобраться с элементами конструкции и с итерациями цикла. Корректно написанный цикл while должен содержать:

- Инициализацию переменной цикла перед началом его выполнения
- Использование текущего значения переменной цикла в его теле
- Изменение (увеличение) текущего значения в теле цикла (часто в конце цикла)

В листинге 2.7 в качестве переменной цикла применяется переменная num. Она инициализируется перед циклом, при определении. Ее значение используется в теле цикла. Оно увеличивается в конце цикла и служит для принятия решения о завершении цикла.

Функции и вызовы функций

Разбиение программы на модули-функции дает возможность распределить работу по реализации ПО между программистами. Группы функций помещаются в разные исходные файлы, и для каждого файла назначается программист. Это позволяет программистам работать параллельно. Очевидно, программист может работать и над несколькими функциями в разных файлах, но несколько человек не могут заниматься одной функцией. Если функция настолько велика, что требует усилий нескольких программистов, ее следует разбить на разные функции.

Другие преимущества использования функций перечислены ниже.

- В вызывающем коде применяются вызовы функций, имена которых могут отражать смысл операций. Это делает исходный код более читабельным, чем при включении в него большого числа операций нижнего уровня.

- Размер исходного (и объектного) кода можно уменьшить: если в разных частях программы выполняются одни и те же операции, надо оформить их как вызов функции, а не повторять в исходном (и объектном) коде как операции нижнего уровня.
- Применение стандартных библиотек (и включение в библиотеки проекта специфических для проекта функций) увеличивает возможности повторного использования кода в одном или нескольких проектах.

Если это делается корректно, разбиение программы на отдельные функции изменяет структуру программы, но не ее вывод. При этом качество программы в значительной степени зависит от того, как именно ее разбить на модули. Независимые функции упрощают понимание и сопровождение программы.

Рассмотрим различные реализации программы, представленной в листинге 2.1. Поскольку эта программа невелика, примеры не демонстрируют преимуществ, связанных с удобством чтения, размером программы и ее повторным использованием. Между тем они позволят показать синтаксис и семантику (смысл) применения функций.

Первое изменение (см. листинг 2.8) включает в себя вывод начального приветствия — это делает отдельная функция с именем `displayInitialGreeting()`. Данная функция вызывается из функции `main()`. Следовательно, `main()` — это клиент, а сама функция является для `main()` сервером.

Листинг 2.8. Первая программа C++ с одной функцией-сервером

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.1415926;

void displayInitialGreeting()                          // заголовок функции
{
  cout << "Добро пожаловать в мир C++!" << endl;       // тело функции
}                                                      // конец блока функции

int main(void)
{
  double x=PI, y=1, z;
  displayInitialGreeting();                            // вызов функции
  z = y + 1;
  y = pow(x,z);
//  cout << "В этом мире pi в квадрате равно " << y << endl;
//  cout << "Приятного дня!" << endl;
  return 0;
}                                                      // конец блока функции
```

Конечно `displayInitialGreeting();` — довольно глупая функция. Она содержит только один оператор. Тем не менее каждая такая функция демонстрирует, что применение функций C++ требует координации трех элементов программы:

- Заголовка функции
- Тела функции
- Вызова функции

Заголовок функции определяет ее интерфейс: тип возвращаемого значения, имя функции, список параметров (в скобках) с типами и имена формальных параметров. Если функция не использует параметров, то список параметров в скобках будет пустым. Если функция не возвращает значения, то возвращается тип `void`.

Это название (void — пусто) описывает смысл обработки и следует популярным соглашениям, предусматривающим комбинирование поясняющих действие глаголов (display — вывод на экран) и существительного, указывающего объект действия (InitialGreeting — начальное приветствие). В соответствии с популярными соглашениями по программированию первое слово функции записывается в нижнем регистре, а первые буквы других слов — с буквы в верхнем регистре.

Как видно, функция displayInitialGreeting() не возвращает никакого значения (возвращает тип void). Вот почему оператор возврата значения (return) в ней не требуется. Если необходимо, его можно включить в функцию, но без возвращаемого значения. Эта функция также не имеет параметров (список параметров пуст). Круглые скобки в заголовке все равно указываются. Чтобы показать отсутствие параметров, можно использовать ключевое слово void:

```
void displayInitialGreeting()                    // заголовок функции
{
  cout << "Добро пожаловать в мир C++!" << endl;      // тело функции
  return;                                        // избегайте ненужного кода
  }
```

Тело функции — это последовательность операторов в фигурных скобках. Таким образом, тело функции представляет собой операторный блок (составной оператор). Каждый оператор завершается точкой с запятой, но сам блок не имеет завершающей точки с запятой. Если нужно, тело функции может содержать определения и объявления переменных, необходимых для вычислений в ее теле. Каждое тело функции имеет собственное пространство имен (область действия). Это означает, что определенные в этой функции переменные (локальные) не будут конфликтовать с именами других переменных, определенных в других функциях. Разработчику функции не требуется координировать эти имена с остальными.

Подобно языку C, в C++ не допускается вложенность определений функций. Следовательно, имена функций являются в программе глобальными и обязаны быть уникальными. Разработчику функции нужно согласовывать ее название с другими проектировщиками, независимо от того, вызывают они ее или нет.

В функции displayInitialGreeting() никаких локальных переменных не определяется. Чтобы подчеркнуть границы тела функции, открывающая и закрывающая фигурные скобки располагаются на отдельных строках. Некоторые полагают, что это увеличивает размер программы по вертикали, не давая никаких удобств при чтении, и не выделяют для фигурных скобок отдельных строк. Тем не менее они оставляют пустые строки между функциями:

```
void displayInitialGreeting()                    // заголовок функции
{ cout << "Добро пожаловать в мир C++!" << endl; }
                                                 // тело функции
```

Третий элемент, касающийся использования функции, ее вызов, состоит из имени функции и списка фактических параметров в круглых скобках. Если аргументов нет, скобки все равно нужны. В отличие от заголовка функции, ключевое слово void в вызове функции использоваться не может:

```
displayInitialGreeting(void);                    // некорректный вызов функции
```

Это еще одна унаследованная из C и удивительная черта. Для создателей языка C простота никогда не была приоритетом. Зачем запоминать, что void можно использовать в заголовке функции, но не в вызове? Они сбросили со счетов тот факт, что накопление таких особенностей только путает программистов.

На самом деле эксперт по языку C знает ответ на этот вопрос и, вероятно, назовет данное решение простым. Но это не так. Я вел семинары по C++ среди большого числа программистов на C. Многие их них — хорошие профессионалы,

но нечасто можно было услышать от них верный ответ. По крайней мере, для этого требовались многочисленные намеки. Вы узнаете ответ, когда лучше освоите язык.

При вызове функции вызывающая функция (клиент) приостанавливает свое выполнение и передает управление вызываемой функции (серверу). Ее операторы выполняются последовательно. Когда достигается закрывающая скобка в теле функции-сервера, эта функция завершает работу и управление возвращается вызывающей функции (вот почему завершение функции называется return — возврат). Затем функция-клиент возобновляет выполнение.

Теперь рассмотрим функцию с параметрами. В листинге 2.9 показана еще одна версия первой программы C++. Она содержит функцию displayResults(), реализующую операции локального блока из предыдущей версии. Функции передается параметр типа double, и она выводит его на экране с дополнительными сообщениями.

Листинг 2.9. Первая программа C++ с двумя функциями-серверами

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.1415926;

void displayInitialGreeting()                          // заголовок функции
{ cout << "Добро пожаловать в мир C++!" << endl; }     // тело функции

void displayResults(double y)                          // заголовок функции
{ cout << "В этом мире pi в квадрате равно " << y << endl;
  cout << "Приятного дня!" << endl;}                   // тело функции

int main(void)
{
  double x=PI, y=1, z;
  displayInitialGreeting();                            // вызов функции
  z = y + 1;
  y = pow(x,z);
  displayResults(y);                                   // еще один вызов функции
  return 0;
    }
```

Функция displayResults() не возвращает никакого значения вызывающей функции (у нее возвращаемый тип void), но список параметров не пуст. Как видно, определение параметра аналогично определению переменных. Программист выбирает имя параметра и задает его тип. Результат этого определения на этапе выполнения аналогичен результатам определения переменной: при вызове функции для параметра соответствующего типа выделяется память. Адрес данной области памяти используется при упоминании имени параметра в теле функции (например, ссылка на переменную y в первом операторе cout). Значение параметра инициализируется значением фактического аргумента в вызове функции.

В вызове функции указываются ее имя и список фактических аргументов в круглых скобках. Здесь в списке только один аргумент (его имя совпадает с именем формального параметра, но это совпадение). Заметим, что фактический аргумент — это выражение, а не определение переменной. Следующая форма вызова функции будет некорректной:

```
displayResults(double y);              // ошибка
```

Далее рассмотрим функцию, возвращающую отличный от void результат и имеющую несколько параметров. Листинг 2.10 содержит еще одну функцию для первой программы C++ (на самом деле это можно сделать несколькими способами). Здесь введена функция computeSquare() с двумя параметрами типа double. Функция возвращает значение типа double.

Листинг 2.10. Первая программа C++ с тремя функциями-серверами

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.1415926;

void displayInitialGreeting()                          // возвращает тип void
{ cout << "Добро пожаловать в мир C++!" << endl; }

double computeSquare(double x, double y)               // возвращает непустой результат
{ double z;                                            // локальная переменная
  z = y + 1;
  y = pow(x,z);
  return y; }                                          // обязательный оператор возврата

void displayResults(double y)                          // заголовок функции
{ cout << "В этом мире pi в квадрате равно " << y << endl;
  cout << "Приятного дня!" << endl;}                   // тело функции

int main(void)
{
  double x=PI, y=1;
  displayInitialGreeting();                            // вызов функции
  y = computeSquare(x,y);                              // еще один вызов функции
  displayResults(y);                                   // и еще вызов
  return 0;
    }
```

Отметим, что на типы параметров и возвращаемые значения никаких ограничений нет. В этом примере они все имеют тип double просто по совпадению.

Каждый параметр задается своим типом и именем. Это во многом напоминает определение переменной. Если параметры имеют один тип, то их нужно описывать отдельно. Параметры нельзя перечислить через запятые, подобно определению переменных. В следующей функции список параметров некорректен:

```
double computeSquare(double x, y)          // синтаксическая ошибка
{ double z;
  z = y + 1;
  y = pow(x,z);
  return y; }
```



Внимание Параметры и возвращаемые значения могут быть одного типа. Тип каждого параметра нужно указывать отдельно, даже если он совпадает.

Так как заголовок функции computeSquare() задает непустое возвращаемое значение (в данном случае double), тело функции должно содержать оператор return. Для этого используется ключевое слово return, а в качестве аргумента указывается double. Некоторые программисты помещают возвращаемое значение в круглые скобки, но в этом нет необходимости. В теле функции computeSquare()

определяется локальная переменная z, ей присваивается значение, вычисляется значение переменной y, и результат возвращается вызывающей функции. (См. главу 5.) Когда код клиента вызывает данную функцию, он передает ей значения аргументов и эти значения используются внутри функции. Поскольку функция возвращает значение, его можно включать в коде клиента в выражения точно так же, как обычное значение данного типа (в нашем случае double):

```
y = pow(x,z);
```

Нужно различать термины параметр и аргумент. *Параметр* — переменная, определенная в заголовке функции и используемая в ее теле, а *аргумент* — переменная, определенная в функции-клиенте и используемая в вызове функции. Часто говорят о формальных и фактических аргументах.

В этом примере в целях простоты для формальных параметров и соответствующих фактических аргументов функции computeSquare() используются одни и те же имена. В реальной жизни такое случается редко. В крупной программе функцию-клиент разрабатывает один программист, а функцию-сервер — другой. Им нет никакой необходимости координировать имена параметров. Программисту, работающему над клиентом, достаточно знать типы параметров и возвращаемых значений.

Часто функция-сервер разрабатывается раньше, чем функция-клиент. Она помещается в библиотеку, а ее исходный код недоступен. Но, даже если он и доступен, нет нужды обременять программиста, занимающегося клиентом, изучением кода сервера и имен формальных параметров. Имена фактических аргументов никак не связаны с именами формальных параметров. Вот еще одна версия функции computeSquare(), которую можно использовать в листинге 2.10. От этого ничего не меняется, что подчеркивает данную независимость:

```
double computeSquare(double base, double exponent)
{ double power = exponent + 1;                    // локальная переменная
  return pow(base,power); }                       // оператор возврата
```

Во всех этих примерах определения функций-серверов размещаются перед вызовами функций в клиенте. Весьма похоже на ситуацию с обычными переменными: определение лексически предшествует использованию переменной. Что произойдет, если разместить функции в исходном коде в другом порядке? Здесь опять вступают в действия ограничения C++, унаследованные от C: если определение функции не предшествует лексически вызову функции, то компилятор его не разглядит и сообщит, что не знает идентификатор displayInitialGreeting() и др. Некорректная версия программы приведена в листинге 2.11.

Листинг 2.11. Некорректная программа C++ с функциями, следующими за вызовами

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.1415926;

int main(void)
{
  double x=PI, y=1;
  displayInitialGreeting();          // синтаксическая ошибка
  y = computeSquare(x,y);            // еще одна синтаксическая ошибка
  displayResults(y);                 // и еще ошибка
  return 0;
    }
```

```
void displayInitialGreeting()                          // определение функции
{ cout << "Добро пожаловать в мир C++!" << endl; }     // тело

double computeSquare(double base, double exponent)
{ double power = exponent + 1;                         // локальная переменная
  return = pow(base,power); }                          // оператор возврата

void displayResults(double y)                          // заголовок функции
{ cout << "В этом мире pi в квадрате равно " << y << endl;
  cout << "Приятного дня!" << endl;}                   // тело функции
```

Компилятор пытается дать полезную информацию о функциях и элементах, требующих переопределения:

```
Compiling...
c:\data\ch02.cpp
c:\data\ch02.cpp(7) : error C2065: 'displayInitialGreeting' : undeclared identifier
c:\data\ch02.cpp(7) : error C2064: term does not evaluate to a function
c:\data\ch02.cpp(8) : error C2065: 'computeSquare' : undeclared identifier
c:\data\ch02.cpp(8) : error C2064: term does not evaluate to a function
c:\data\ch02.cpp(9) : error C2065: 'displayResults' : undeclared identifier
c:\data\ch02.cpp(9) : error C2064: term does not evaluate to a function
c:\data\ch02.cpp(13) : error C2371: 'displayInitialGreeting' : redefinition;
    different basic types
c:\data\ch02.cpp(17) : error C2371: 'computeSquare' : redefinition;
    different basic types
c:\data\ch02.cpp(23) : error C2371: 'displayResults' : redefinition;
    different basic types
DEMO.EXE - 9 error(s), 0 warning(s)
```

Конкретный вид данных сообщений зависит от типа применяемого компилятора, но это не важно. Обычно начинающий программист сначала не верит: "Как идентификатор может быть не определен? Он определен — вот здесь. Эй, компилятор, ты ослеп?" Компилятор не ослеп. Это все тот же компромисс между интересами разработчиков компилятора и интересами пользователей. Пользователь компилятора упустил из виду главное правило: все, с чем работает программа, предварительно должно быть определено. Речь идет действительно *обо всем*: включая имена переменных и функций.

Цель применения такого требования к вызовам функций очевидна: важно, чтобы компилятор проверил правильность написания имени функции, корректность числа аргументов и их типов. Разумеется, сообщения об ошибках могли бы быть более конкретны. И конечно, такую проверку можно было бы сделать, не требуя чего-то от программиста. Второй проход по исходному коду (как делается в некоторых старых компиляторах) полностью устранил бы проблему.

В C++ программисты решают данную проблему с помощью прототипов функции. Прототип функции имеет ту же синтаксическую форму, что и заголовок функции. Единственное отличие в том, что прототип должен завершаться точкой с запятой. На размещение прототипов никаких ограничений не налагается. Они лишь должны лексически предшествовать первому вызову функции. В соответствии с общепринятым соглашением, прототипы функции размещаются в начале исходного файла с функциями-клиентами.

Соотношение между определениями функций и прототипами функций — это обобщение соотношений между определениями и описаниями переменных. Прототип функции есть описание функции, и его можно повторять любое число раз. Определение функции может встречаться в программе только один раз.

При применении прототипов функций нет необходимости включать определения в тот же файл, где находится функция-клиент. Наличие прототипов вполне достаточно для компилятора. В листинге 2.12 показана корректная версия листинга 2.11 с прототипами функции-сервера в начале файла и перенесенными в другое место определениями функций.

Листинг 2.12. Корректная программа C++ с прототипами, предшествующими вызовам функций

```
const double PI = 3.1415926;

void displayInitialGreeting()                    // прототипы функции
double computeSquare(double x, double y);
void displayResults(double y);

int main(void)
{
   double x=PI, y=1;
   displayInitialGreeting();                     // вызовы функции
   y = computeSquare(x,y);
   displayResults(y);
   return 0;
}                                                // конец блока функции
```

Даже этот небольшой пример показывает преимущество использования функций. Из функции `main()` убраны детали операций — здесь видно, *что* делается, а не *как*. Чтобы ознакомиться с деталями, программисту, сопровождающему программу, достаточно обратиться к исходному коду функций-серверов в разных файлах. Поскольку каждая функция находится в отдельном файле, его внимание не будет рассеиваться на ненужные детали. Программисту не надо даже знать имен исходных файлов с функциями-серверами. Между тем объектный код этих функций должен компоноваться с объектным кодом `main()`. Исчезли и файлы `#include`. Они включены в исходный код функций-серверов, вызывающих библиотечные функции. Программисту, занимающемуся клиентом, не требуется знать, какой сервис использует сервер.

Если к серверной функции обращаются разные функции-клиенты в разных файлах, то в каждый такой файл должен включаться прототип данной функции. Часто программисты помещают прототипы в отдельные заголовочные файлы и включают их в файлы при реализации функций-клиентов. Разница между включением стандартных библиотечных файлов типа `iostream` и определяемых программистом заголовочных файлов состоит в использовании имен маршрутов и двойных кавычек. В листинге 2.13 прототипы перемещены в файл по маршруту `c:\data\cppbook\ch02\diisplay.h`:

Листинг 2.13. Программа C++ с прототипами в отдельном файле

```
const double PI = 3.1415926;
#include "c:\data\cppbook\ch02\display.h"        // прототипы

int main(void)
{
   double x=PI, y=1;
   displayInitialGreeting();                     // вызовы функций
   y = computeSquare(x,y);
   displayResults(y);
   return 0;
}                                                // конец блока функции
```

Использовать имена параметров в прототипах функций необязательно. Компилятор не контролирует по ним корректности вызовов функций: анализируются только типы аргументов. Эти имена могут быть полезными для документирования. Если имена параметров не используются, то файл c:\data\cppbook\ch02\diisplay.h может выглядеть так:

```
void displayInitialGreeting();
double computeSquare(double, double);                    // нет имен параметров
void displayResults(double);
```

Классы

Теперь приступим к описанию синтаксиса, позволяющего комбинировать функции с данными в составной тип данных. В C++ для этого предусматриваются ключевые слова struct и class, а также правила слияния отдельных компонентов в единое целое. В определении класса задаются типы и имена элементов данных (полей) и заголовки или тела функций-членов (обращающихся к полям данных). Имя класса может использоваться в коде клиента как имя типа. Это означает, что можно определять переменные данного типа (объекты), передавать их функциям как параметры и т. д.

Представим время дня как комбинацию часов и минут. Желательно, чтобы объект мог хранить данные времени и выводить сохраняемые значения. Код клиента может запрашивать время в разном формате (18:45 или 6:45 P.M.).

Для этого создадим описание класса с двумя целочисленными полями данных: hours и munutes, к которым можно обращаться извне класса. Сначала описывается композиция объектов класса (экземпляры и переменные), которые позднее будут определяться и использоваться в программе:

```
struct TimeOfDay          // используется ключевое слово struct
  {
int hours;                // один компонент данных: целое
int minutes;              // другой компонент данных: целое
  } ;
```

В соответствии с общим соглашением, имена классов записываются с буквы в верхнем регистре, а каждая составляющая имени также начинается с буквы в верхнем регистре. Открывающая и закрывающая фигурные скобки отмечают область действия класса — границы, отделяющие содержимое класса от того, что находится вне его. В отличие от других случаев использования области действия в фигурных скобках, здесь точка с запятой после закрывающей скобки обязательна. Так что не думайте, будто в C++ все очень просто.

Определения полей данных в классе аналогичны определениям переменных. Они ассоциируют тип и имя поля. Когда создается объект (экземпляр переменной) класса TimeOfDay, для каждого поля в соответствии с его типом выделяется память.

```
TimeOfDay time1, time2;          // распределяются два объекта
```

Если поля одного типа, то можно использовать только одно имя типа, а поля разделять запятыми. Это аналогично определениям переменных встроенных типов C++.

```
struct TimeOfDay
  {
    int hours, minutes;          // другой элемент данных: целое
  } ;
```

Опять же можно применять этот синтаксис для определения переменных и полей данных класса, но не для параметров функции.

Поля данных должны быть закрытыми. Они недоступны вне класса. Далее понадобятся функции-члены (методы), обращающиеся к полям данных от имени клиента. Эти функции должны быть общедоступными, чтобы в коде клиента их можно было использовать для установки и вывода времени в двух форматах. Например, реализуем функции `setTime()`, `displayMilitaryTime()` и `displayTime()`. Функция `setTime()` должна иметь два параметра (часы и минуты). Другие функции параметров не имеют. Они просто выводят на экран значения, хранящиеся в объекте. В листинге 2.14 представлен синтаксис определения класса с элементами данных и функциями-членами.

Листинг 2.14. Первый класс C++, комбинирующий функции и данные

```
#include <iostream>
using namespace std;
class TimeOfDay {                          // используется ключевое слово class

  private:                                 // ключевое слово private делает данные скрытыми

    int hours, minutes;

  public

    void setTime(int hrs, int min)
         { hours = hrs; minutes = min; }

    void displayMilitaryTime(void)
         { cont <<hours<<":"<<minutes; }

    void displayTime(void)
         { if (hours > 12)
            cout << hours-12<<":"<<minutes<<"P.M.";
           else
            cout << hours <<";"<<minutes<<A.M."; }
} ;
```

Функция `setTime()` копирует значения параметров в поля данных объекта. Функция `displayMilitaryTime()` выводит часы и минуты. Функция `displayTime()` проверяет, не превышает ли значение `hours` (часы) 12. Если да, то она вычитает из `hours` 12 и выводит разницу с минутами и P.M. (после полудня). Если нет, `hours` выводится с `minutes` и A.M. (до полудня). В данном примере используется библиотека C++ `iostream`.

Эти функции находятся в самом классе, а следовательно, могут работать с его данными без ограничений — устанавливать значения и обращаться к ним в соответствии с целями клиента. Собственно, они и нужны для клиентского кода, а не для класса. Вот почему функции-члены определены как общедоступные (public). Их можно вызывать в клиентском коде. Для этого клиентский код определяет объекты класса с помощью стандартного синтаксиса определения переменных C++: клиент ассоциирует имя типа (`TimeOfDay`) и имена переменных (например, `time1`, `time2`).

Определение класса сохранено в файле `c:\data\cppbook\ch02\time.h`. Чтобы можно было использовать этот класс, нужно включить его в данный заголовочный файл. В противном случае компилятор сообщит, что имя `TimeOfDay` не определено. В листинге 2.15 показан пример исходного кода клиента, в котором определяются объекты `TimeOfDay`, анализируется возвращаемое функциями класса значение и форматируется вывод. (Здесь включается заголовочный файл.)

Листинг 2.15. Код клиента для первого класса C++, комбинирующего функции и данные

```
#include <iostream>
using namespace std;
#include "c:\data\cppbook\ch02\time.h"

int main(void)
{
  TimeOfDay time1, time2;                  // экземпляр класса
  int hours = 19, minutes = 15;            // целочисленные переменные
  time1.setTime(7,35);
  time2.setTime(hours, minutes);           // инициализация объектов
  cout << "Первое время: ";
  time1.displayMilitaryTime();             // сообщение первому объекту
  cout << endl << "Первое время: ";
  time1.displayTime();
  cout <<endl<< "Второе время: ";
  time2.displayMilitaryTime();             // сообщение второму объекту
  cout << endl << "Второе время: ";
  time2.displayTime();
  return(0);
}
```

Переменные `time1` и `time2` имеют тип `TimeOfDay`. Они определены аналогично переменным встроенного типа. По существу, определение класса расширяет доступный в C++ набор типов и добавляет в него наш тип `TimeOfDay`. Несмотря на этот скромный вклад, классы — весьма мощное средство в C++. Они открывают огромные возможности и делают язык более мощным. Связь между кодом клиента и кодом класса показана на рис. 2.6. Когда клиенту нужно обращаться к закрытым частям класса (для установки значения времени или его получения), он не обращается непосредственно к данным (пунктирная линия на рисунке), а вызывает функции-члены `setTime()`, `displayTime()`, `displayMilitaryTime()`, сплошные линии на рис. 2.6. Эти функции обращаются к закрытым данным класса от имени клиента.

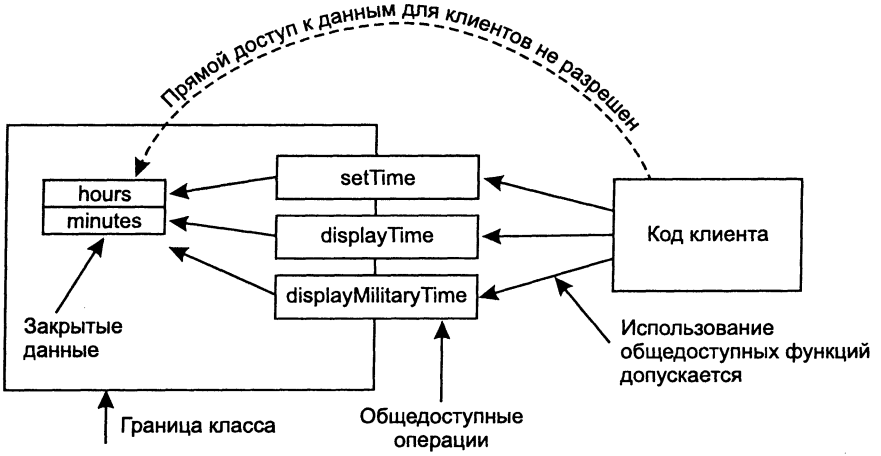


Рис. 2.6. *Код клиента использует функции доступа, а не обращается к данным непосредственно*

Вызов функции-члена называется сообщением объекту. Обратите внимание на синтаксис сообщения: когда вызывается функция-член, следует указать, с каким объектом функция должна работать (для этого используется операция точки).

```
Первое время:    7:35
Первое время:    7:35A.M.
Второе время:   19:15
Второе время:    7:15A.M.
```

Рис. 2.7. *Вывод функций доступа TimeOfDay*

Если целевым объектом является time1, то выводятся поля данных hours и minutes объекта time1. Если целевой объект — time2, то выводятся значения из time2. Вывод программы показан на рис. 2.7.

В отличие от других, показанных ранее функций, функции-члены нельзя вызывать просто по имени (с указанием аргументов, если необходимо):

```
displayTime();                    // синтаксическая ошибка
```

Какое время здесь нужно вывести? Из какого объекта? Ничего этого не задается, и компилятор не знает, что программист имеет в виду. Вот почему такой вызов функции даст ошибку. Нужно указать цель сообщения (получателя), но данная цель должна быть того же типа, что и вызываемая компонентная функция. Переменную другого типа в качестве цели использовать нельзя. Например, следующая запись некорректна:

```
hours.displayTime();              // синтаксическая ошибка
```

Переменная hours имеет тип int. С целочисленными выражениями можно выполнять такие операции, как вычитание, сложение и пр., но не операцию displayTime(). Именно поэтому оператор указывает на ошибку. Цель должна иметь верный тип.

Расширение языка C++ с помощью классов поддерживается такими средствами, как композиция классов и наследование. Можно было бы дать простые примеры композиции и наследования, но это сделало бы введение слишком длинным. Оно и без того достигло цели, познакомив читателя с C++. Это вполне позволяет писать небольшие программы — не очень элегантные, но надежные и работающие. Еще важнее, что первое короткое знакомство послужит основой для дальнейшего изучения C++.

Прежде чем приступать к более подробному ознакомлению с C++, следует рассмотреть вопросы компоновки и выполнения программ C++.

Применение инструментальных средств разработки программного обеспечения

Как уже упоминалось, для создания исходного кода C++ используется текстовый редактор. Полученный исходный код обрабатывается компилятором C++. В случае синтаксических ошибок генерируются соответствующие сообщения, а если ошибок нет, можно запускать программу. В этом разделе рассматриваются более развитые инструментальные средства, которые можно применять для создания и выполнения программ C++, поясняется роль каждого инструментального средства и способы их наиболее эффективного использования. Естественно, специфика зависит от того, поддерживает ли этот инструментарий режим командной строки (как, например, компилятор GNU в UNIX) или представляет собой интегрированную среду разработки (как Microsoft IDE в Windows).

Многие среды разработки требуют (или предполагают), что в дополнение к сохранению исходных файлов добавляются к проекту файлы (проект может содержать несколько исходных файлов). Файлы проекта, интерпретируемые как единое целое, можно применять для сбора информации о программе, которая пригодится для ее анализа и отладки. Кроме того, поставщики интегрированных сред разработки ПО часто предусматривают возможность генерации многофайловой базовой структуры, позволяющей добавлять код приложения или класса к "скелетам" функций и классов, генерируемых самим инструментарием. Для опытного программиста это может быть очень полезным, но при изучении языка будет только мешать. Вместо того чтобы сосредоточиться на конкретных темах языка, начинающий программист будет изучать сгенерированный для него исходный код, а в нем могут применяться незнакомые ему средства.



Советуем Чтобы избежать лишних сложностей и упростить задачи управления проектом, во время экспериментирования с языком полезно придерживаться однофайловых программ. Некоторые интегрированные среды создают проект по умолчанию, хотите вы того, или нет. Остается с этим смириться, но при изучении C++ лучше избегать использования сгенерированного средой программного кода, что упростит вам жизнь.

Полезно сохранять свои файлы после редактирования и перед тестированием программы. Если не сделать этого, а программа содержит ошибки этапа выполнения, способные привести к аварийному завершению работы системы, то введенные изменения будут потеряны. Сегодня лишь немногие записывают операторы на бумаге, прежде чем набрать их с клавиатуры, и в результате аварийного завершения системы можно потерять значительную часть работы. Сохранение файла на диске (под другим именем) перед компиляцией сведет опасность к минимуму. Некоторые программы-редакторы выводят на экран информацию о статусе файла, которая может оказаться полезной. Если код изменен, а файл еще не сохранен, редактор помечает имя файла в окне редактирования звездочкой. После сохранения файла звездочка (или сообщение) исчезает.

Другие интегрированные среды разработки не доверяют здравому смыслу программиста и сохраняют исходный файл, как только получают команду на компиляцию, даже если изменения носят "экспериментальный" характер и могут подлежать отмене.

Если исходный файл не содержит синтаксических ошибок, компилятор генерирует объектный файл, имя которого совпадает с именем исходного файла. В зависимости от системы, имя объектного файла получает расширение .o или .obj. Если код содержит синтаксические ошибки, объектный файл не создается. Компилятор выводит сообщения об ошибках: показывает, на какой строке они найдены и что это за ошибки. В старых инструментальных средствах приходилось считать строки исходного кода или полагаться на номера строк, выводимые редактором. Новый инструментарий устраняет необходимость трассировки номеров строк, хотя и указывается, в какой строке произошла ошибка. Достаточно щелкнуть мышью сообщение об ошибке, и курсор позиционируется в окне редактора в том месте, где эта ошибка найдена. Очень полезное средство.

Если освоиться с языком, то сообщения об ошибках станут ясными и полезными. По крайней мере, некоторые из них. До той же поры лучше полагаться на правило: никогда не понимать сообщения компилятора буквально, не анализировать их, или, по крайней мере, остановиться при первой же трудности. Не упорствуйте. Не стоит.

Как минимум 90% сообщений об ошибках ничего не говорят начинающим. Есть ли исключения? Конечно. Около 10% сообщений об ошибках вполне понятны, но это как раз те случаи, когда сообщение становится понятным, если посмотреть на строку исходного кода, где содержится ошибка. Вот почему лучше не тратить время, энергию и нервы на сообщения об ошибках. Если ближе познакомиться с их терминологией, они будут более полезны, подскажут, в каком направлении двигаться. На первых же этапах изучения языка чтение сообщений об ошибках принесет больше вреда, чем пользы.

Причина в том, что разработчики компилятора хотели дать программистам наиболее полезную информацию и детальный анализ ситуации. Не удивительно, что в их сообщениях используется сложная терминология C++, незнакомая новичку. Порой бывает так, что проблема вызвана ошибками совсем иного рода, и у компилятора нет ни времени, ни "знаний" на детальный анализ и корректное указание причины ошибки. Таким образом, сообщение с техническими и специфическими терминами часто имеет очень мало общего с фактической ошибкой.

Честно говоря, не стоит рассчитывать на существенную помощь компилятора с его сообщениями об ошибках. Лучше полагаться на собственный разум. Единственная надежная информация в сообщении об ошибке — это место, где она произошла. Все остальное не всегда корректно. И даже такая информация бывает некорректной. Иногда источник ошибки — предыдущая строка. Так что, когда получите сообщение об ошибке (а без этого вряд ли обойдется), не концентрируйтесь на указанной строке — проверьте и предыдущую.

Часто компилятор выдает несколько сообщений об ошибках. Многие программисты, сомневаясь в причине, начинают с первого сообщения, потом анализируют второе, третье. Им кажется, что они увидят в них больше смысла.

Не стоит идти таким путем. Не анализируйте второе сообщение, если не поняли первого. Исправив первую ошибку, не беритесь сразу за вторую. Это неправильно. Раньше компиляция занимала несколько минут или даже часов. В таком случае имело смысл потратить время на анализ сообщений компилятора (даже спонтанных) и устранить максимальное число ошибок.

В наши дни компиляция выполняется за секунды. Это быстро и недорого, так что не тратьте время на анализ второй и третьей ошибок. Они могут появиться не из-за реальных ошибок, а потому, что компилятор не уследил за ходом программы из-за первой ошибки. Время программиста стоит дороже времени компилятора. Более опытные программисты отличают ложные ошибки от реальных. Начинающим же лучше исправлять первую ошибку и перекомпилировать программу.

Ниже приведена версия листинга 2.1 с одной небольшой ошибкой. Компилятор генерирует три сообщения об ошибках с номерами строк.

```
#include <cmath>                                        // директива препроцессора
#include <iostream>                                     // директива препроцессора
using namespace std;                                    // директива компилятора
const double PI = 3.1415926;                            // определение константы

int main(void)                                          // функция возвращает целое значение
{
   double x=PI, y=1, z,                                 // определение переменных
   cout << "Добро пожаловать в мир C++!" << endl;       // вызов функции
   z = y + 1;                                           // операция присваивания
   y = pow(x, z);                                       // вызов функции
   cout << "В этом мире pi в квадрате равно " << y << endl;
   cout << "Приятного дня!" << endl;
   return 0;                                            // оператор возврата
}                                                       // конец блока функции
```

```
Compiling...
ch02.cpp
c:\Data\ch02.cpp(7) : error C2143: syntax error : missing ';' before '<<'
c:\Data\ch02.cpp(7) : error C2143: syntax error : missing ';' before '<<'
c:\Data\ch02.cpp(10) : error C2296: '<<' : illegal, left operand has type 'double'
c:\Data\ch02.cpp(10) : error C2297: '<<' : illegal, right operand has type 'char [29]'
c:\Data\ch02.cpp(11) : error C2296: '<<' : illegal, left operand has type 'double'
c:\Data\ch02.cpp(11) : error C2297: '<<' : illegal, right operand has type 'char [17]'
Error executing c1.exe.

ch02.exe - 6 error, 0 warning(s)
```

Вся эта интеллектуальная деятельность компилятора совершенно напрасна. Можно потратить часы, пытаясь найти истинную причину и пробуя добавить точку с запятой перед << в строке 7, или рыться в справочнике, чтобы понять, что не так с типами в строках 10 и 11. Все это ложный путь. Строки 7, 10 и 11 не содержат ошибок — не важно, что говорит компилятор. Ошибка в строке 5. К примеру, программист просто нажал запятую вместо точки с запятой.

И еще одно замечание относительно сообщений компилятора. Программисты весьма часто делают такого рода ошибки. Они простые, незаметные и неожиданные. Учитывая вводящие в заблуждение сообщения компилятора, найти их бывает непросто. Нередко под подозрение попадают совсем другие операторы, особенно, если их синтаксис не очень хорошо понятен. Программист может приступить к исправлению этих операторов. Они становятся неверными, что лишь увеличивает число сообщений об ошибках. Программист анализирует их, находит еще что-то, пытается исправить, но опять усугубляет дело и т. д. Потратив много времени (и нервов), он, наконец, находит ошибку с мыслью: "Как можно было сделать такую глупую ошибку? Как я мог ее не увидеть? Как плохо быть таким тупым..."

Так что совет "не обращать внимания на сообщения компилятора об ошибках" нужно понимать буквально. Не следует оценивать себя по этим ошибкам. В конце концов они будут найдены и исправлены, но важно то, сколько времени на это уйдет. Важна уверенность в своих знаниях. Излишняя самокритичность ее подорвет. Настанет момент, когда вы заслужите одобрительного похлопывания по плечу: ведь вы устранили ускользающую от внимания проблему. Не вините себя. Развивайте чувство уверенности в своих силах и способностях.

Иногда компилятор генерирует предупреждающие сообщения. Когда наберетесь опыта, лучше анализировать их аналогично сообщениям об ошибках, ведь предупреждения часто отражают реальные ошибки. Со временем лучше перестать обращать на них внимание. Они порой вводят в заблуждение. В то же время, предупреждения не препятствуют генерации объектного кода и отладки, т. е. можно протестировать исходный код на этапе выполнения. Позднее следует разобраться в предупреждениях и избавиться от них.

Следующий шаг после успешной компиляции — компоновка (в некоторых инструментальных средствах это называется building — построение программы). Если программа состоит из нескольких исходных файлов, каждый файл может (и должен) в процесс разработки компилироваться отдельно. Когда исходный код содержит описания идентификаторов (переменных или функций), они часто определяются в другом файле, и у компилятора возникает проблема — он не знает адресов этих идентификаторов, так как обрабатывает файлы поочередно. Компоновщик проходит все объектные файлы и разрешает внешние ссылки на идентификаторы, определенные в других файлах.

При компоновке к объектному добавляется другой скомпилированный код из некоторых библиотек C++. Хотя библиотечный исходный код обычно доступен, он не компилируется при каждом вызове `pow()`, `operator`, `<<` или другой библиотечной функции. Такие функции компилируются заранее, и компоновщик разрешает эти внешние ссылки так же, как другие.

Результат этапа компоновки — выполняемый файл программы. Если это программа из одного файла, то имя выполняемого файла будет совпадать с именем исходного. Для многофайловой программы именем выполняемого файла будет имя проекта (обычно программист может выбрать любое имя). Расширением, как правило, является `.exe`.

Ошибки компоновки встречаются нечасто. Они происходят, как правило, из-за неверного имени функций или некорректных операций с проектом.

Выполняемый файл можно запускать. Большинство интегрированных средств разработки (IDE) позволяют выполнять программу в режиме отладки. Замечательно, но лучше отладчик не использовать. На первых этапах освоения C++ вы будете заняты изучением языка, редактора, компилятора и компоновщика — без этого не обойтись. Изучение отладчика — непростая задача. Оно себя не окупит и не даст заметной экономии времени, по крайней мере, пока вы не начнете писать сложные программы C++. А до той поры достаточно исследования вывода программы (и добавления в нее при необходимости дополнительных операторов вывода).

И еще одно предупреждение. Очень часто, глядя на результаты программы, программист не замечает их некорректности. Трудно понять, почему, но факт. Возможно, это как-то связано с самовнушением, но, кто знает... Как бы то ни было, ошибки этапа выполнения часто пропускают.

Некоторые программисты пытаются избежать этого, записывая предполагаемый вывод еще до запуска программы. Полезный метод, но он не дает полной гарантии. Так что будьте бдительны, проверяя результаты тестов.

Итоги

Итак, вы познакомились с самым важными компонентами языка программирования C++, знаете, что такое директивы препроцессора и как их использовать, как определять переменные и функции, управлять ходом выполнения программы с помощью операторов условия и циклов, умеете комментировать исходный код, не обращать внимания на вводящие в заблуждения сообщения компилятора. И даже имеете представление о том, как определять и использовать класс C++. Замечательно!

Фактически этого достаточно для написания большей части кода C++, который потребуется писать, что весьма напоминает изучение обычного языка. "Упрощенный" английский намного компактнее и легче обычного английского, но даже такая малая часть языка позволяет выразить удивительно много. Возможно, это еще одно проявление правила, согласно которому 80% работы делают 20% принимающих в ней участие.

Но не стоит ограничивать себя этими 20%. Пришло время двигаться дальше и приступить к изучению средств языка, доступных опытному программисту.

Глава 3

Работа с данными и выражениями C++

Темы данной главы

- ✔ Значения и их типы
- ✔ Интегральные типы
- ✔ Типы с плавающей точкой
- ✔ Работа с выражениями C++
- ✔ Смешанные выражения: скрытая опасность
- ✔ Итоги

В данной главе рассказано о том, как работать с данными C++, какие доступны типы данных, какие поддерживаются операции со значениями разных типов, каких недостатков C++ следует опасаться программисту. Как и многое другое, C++ объединяет в себе противоположные черты. Его набор целочисленных типов данных очень мал, разница между существующими типами невелика, а выбор между ними не всегда очевиден. Между тем набор операций в этом языке очень велик. Некоторые операции C++ весьма сложны, другие отличаются необычной записью. Объединяют типы данных и операции проблемы переносимости: не всегда все одинаково работает на разных машинах.

C++ унаследовал из языка C исключительную гибкость преобразования значений из одного типа в другой и их комбинирования в сложные выражения. Давайте посмотрим, что он предлагает.

Значения и их типы

В C++ каждое значение в каждый момент своего существования (при выполнении программы) характеризуется своим типом. Переменные C++ ассоциируются с типами при определении. Тип описывает следующие характеристики значения:

- Размер значений данного типа в памяти компьютера
- Набор допустимых для типа значений (метод интерпретации представляющих значение данного типа последовательностей битов)
- Набор действительных для типа операций

Например, для значений типа int на некоторых машинах выделяется 4 байта, диапазон разрешенных значений составляет от –2 147 483 648 до 2 147 483 647. Набор допустимых операций включает в себя присваивание, сравнение, сдвиги, арифметические операции и некоторые другие. Значения типа TimeOfDay, о котором мы говорили в главе 2, имели размер, вдвое больший, чем int (если компилятор не добавлял места для выравнивания в памяти с целью обеспечения более быстрого доступа). Для TimeOfDay допустимым является набор значений, составляющих любые комбинации значений первого целого (от 0 до 23) и второго целого (от 0 до 59). В число допустимых операций с TimeOfDay входят setTime(), displayTime() и displayMilitaryTime(), присваивания, но не сравнения. Конечно, компоненты TimeOfDay сравнить можно (это целые, и к ним применимы правила int), но сами значения типа TimeOfDay сравнивать нельзя. Следует различать свойства типа и свойства его компонентов. Если в коде клиента нужно сравнить значения TimeOfDay, класс должен поддерживать это с помощью реализации функций isLater() или compareTime(). (Обратите внимание, что здесь снова используется клиент-серверная терминология).

Каждая переменная C++ должна определяться путем спецификации типа ее значений. Кроме того, типы характеризуют значения констант, функций и выражений. Это означает, что можно комбинировать типизированные значения в выражения, где результатом будут другие типизированные значения, использовать их в других выражениях и т. д.

В большинстве случаев тип обозначается идентификатором, т. е. имеет имя (например, TimeOfDay). Такой способ общепринят и вполне естествен, но это не единственный путь определения типа. C++ допускает так называемые анонимные типы, не имеющие конкретных имен. Эти типы не так широко используются.

Для встроенных типов C++ зарезервированы имена: int, char, bool, float, double и void (фактически, это полный список). Тип void обозначает отсутствие значения, которое можно использовать в выражениях. Мы будем применять его, когда требуется указать на невозможность включения значения в выражения. Например, функция computeSquare() в главе 2 возвращает значение, включаемое в выражения, а функцию displayResults() в том же разделе таким образом использовать нельзя, так как она значения не возвращает. Если попытаться сделать это, компилятор даст сообщение об ошибке.

```
int a, b;
a = computeSquare(x, y) * 5;              // допускается в C++
b = displayResults(Pi*Pi) * 5;            // ошибка
```

Другие языки таких специальных "типов" не имеют, так как в них различаются функции (возвращающие значения) и процедуры (не возвращающие значений). C++ наследует синтаксис функций из C, где процедуры и функции едины. Логически отсутствие заданного типа возвращаемого значения можно интерпретировать как отсутствие типа возвращаемого значения. В языке C это не так. Более того, отсутствие спецификации типа в языке C означает целочисленный тип. C++ реализует некий компромисс. Если возвращаемый тип не задается, компилятор не требует, чтобы функция возвращала целый тип (как компилятор C). Он предполагает, что возвращается тип void.

```
displayResults(double y)                                  // в C++ это void
{
  cout << "В этом мире pi в квадрате равно " << y << endl;
  cout << "Приятного дня!" << endl;                       // нет ошибки в C++
  }
```

Если употребить эту функцию как операнд в выражении, C++ предполагает, что используется старое соглашение C, и программист хочет возвращать целое

значение. На этапе выполнения функция displayResults() будет возвращать "мусор". Компилятор, не долго думая, просто "снимает защиту":

```
b = displayResults(PI*PI) * 5;              // не синтаксическая ошибка
```

Если оператор return не указывается, то функция, не возвращающая типа, интерпретируется как функция с целочисленным возвращаемым значением:

```
displayResults(double y)                    // C++ предполагает, что это int
{
  cout << "В этом мире pi в квадрате равно " << y << endl;
  cout << "Приятного дня!" << endl;
  return 0;                                 // не синтаксическая ошибка
  }
```

Если необходимо, код клиента может использовать возвращаемое значение:

```
b = displayResults(PI*PI)*5                 // это допустимо
```

Применение int как заданного по умолчанию типа возвращаемого значения тянется из тех времен, когда большинство функций C разрабатывались для возврата значений и программист мог сэкономить три нажатия клавиши (нетривиальное преимущество). Лучше избегать такой практики. Если функция должна возвращать целое значение, напишите int. А если функция не возвращает значения, обозначьте ее тип как void.



Осторожно! Всегда указывайте возвращаемый функцией тип. Если функция не возвращает значения, напишите void. Не полагайтесь на тип, назначаемый C++ по умолчанию.

Типы, определяемые в программе в дополнение к встроенными типам C++, называются пользовательскими. Мне не нравится эта терминология, так как пользователи не определяют типов. Пользователь — человек или организация, применяющие разработанную систему для достижения каких-то целей. Состав типов и их имена определяет программист. Один из примеров — функция TimeOfDay в главе 2. Вот почему предпочтительнее называть их типами, определяемыми программистом.

Хотя разные типы имеют в C++ разный размер, нет ничего необычного в том, что значения разных типов могут иметь одинаковый размер в памяти. Для разных типов значения отличаются интерпретацией представляющих эти значения битов. Например, битовая последовательность 01000001 интерпретируется как 65, если хранится в целочисленной переменной, или как A, если хранится в переменной символьного типа.

Раньше программисты должны были разбираться в двоичных, восьмеричных, шестнадцатеричных числах, кодах ASCII и EBCDIC, запоминать, сколько будет 2 в 16-й степени (а иногда в 20-й или даже в 32-й), понимать, что такое представление с дополнением до 1 или до 2 от отрицательных чисел. Сегодня большинству из них этого не требуется. Тем не менее оборудование компьютера устроено так, что работает с 8-битовыми байтами. В полуслове — 16 бит, в слове — 32. На некоторых машинах слово имеет размер 16 бит, а двойное слово — 32, поэтому полезно знать хотя бы диапазон значений, которые можно хранить в памяти различного размера.

Итак, 4 бита (одна шестнадцатеричная цифра) могут представлять 16 различных комбинаций. Обычно эти 16 комбинаций присваиваются целым числам от 0 до 15. Аналогично 8 бит могут представлять 256 значений (2 в степени 8). 256 комбинаций присваиваются целым числам от 0 до 255. Как быть, если нужно представлять положительные и отрицательные числа, а не только положительные?

В нашем распоряжении все равно 256 комбинаций. Диапазон от –128 до +128 не подойдет, так как в нем 257 значений, а не 256. Стандартное решение — представлять числа от –128 до +127.

Два байта (16 бит) позволяют представить 65 536 битовых комбинаций (2 в степени 16). Для положительных чисел это диапазон от 0 до 65 535, а для отрицательных — от –32 768 (2 в степени 15) до +32 767 (2 в степени 15 – 1). Аналогично 32 бита (4 байта) могут представлять 4 294 967 296 значений. Для чисел со знаком (signed) четыре байта покрывают диапазон от –2 147 483 648 (2 в степени 31) до +2 147 483 647. Вероятно, это все, что нужно знать о двоичных числах.

Интегральные типы

На машинах любой архитектуры целочисленные типы C++ являются базовыми. Что означает "базовый"? Просто то, что операции с этими значениями на любой платформе выполняются быстрее. Для обозначения такого типа служит ключевое слово int:

```
int cnt;
```

Размер int определяет диапазон значений, доступных для представления (2 в степени, равной числу битов). Сейчас отрасль переходит от 16-разрядной к 32-разрядной архитектуре, но обе они какое-то время будут использоваться одновременно. В большинстве стационарных инсталляций это 32-разрядные платформы, но во встроенных и коммуникационных системах продолжают применяться 16-разрядные процессоры, число подобных систем растет. Микропроцессоры можно встретить теперь в автомобилях, крупной бытовой технике и даже в тостерах. Это означает, что программы, написанные для одной архитектуры, не будут работать точно так же на другой.

Что произойдет, если сохраняемое целое значение не поместится в отведенную память? Ничего особенного. В C++ нет такого понятия как арифметическое переполнение. Хотите прибавить 1 к 32 767 на 16-разрядной машине? Ради бога. Результатом будет –32 768. Хотите прибавить еще единицу? Прибавьте. Получите — 32 767.

Листинг 3.1 Демонстрация целочисленного переполнения

```
#include <limits>
#include <iostream>
using namespace std;

int main(void)
{
  int num = INT_MAX - 2;
  int cnt = 0;
  cout << "Целочисленное переполнение в C++:" endl;
  cout << "Увеличение от " << num << endl;
  while (cnt < 5)
  { num = num + 1;
    cnt = cnt + 1;
    cout << cnt << " " << num << endl; }
  cout << "Спасибо, что побеспокоились о границах диапазона целого" << endl;
  return 0;
    }
```

```
Целочисленное переполнение в C++:
Увеличение от 32765
1   32766
2   32767
3   -32768
4   -32767
5   -32766
Спасибо, что побеспокоились
о границах диапазона целого
```

Рис. 3.1. *Целочисленное переполнение не прерывает выполнения программы — она просто дает некорректные результаты*

В листинге 3.1 показана программа, которая выполнялась на 16-разрядной платформе (это была 32-разрядная машина с 16-разрядным компилятором). Заголовочный файл limit содержит библиотеку констант для зависимых от реализации числовых значений на данной платформе. Константа INT_MAX — одно из таких значений (32 767). В данном примере использовался цикл while (см. главу 2) и библиотека iostream. Вывод программы представлен на рис. 3.1. Переменная num благополучно проходит весь диапазон и начинает принимать отрицательные значения. Каждый элемент оператора cout имеет собственную операцию вывода << (даже разделитель в двойных кавычках между выводимыми значениями cnt и num).

В прежних версиях C++ (и C) для инициализации переменных нельзя было использовать значения, получаемые на этапе выполнения — они должны были вычисляться во время компиляции. Однако всегда можно инициализировать переменные не только конкретным значением, но и выражением (например, INT_MAX-2 в листинге 3.1). В современном варианте C++ инициализация выражений может быть достаточно сложной и даже содержать значения, возвращаемые при выполнении функций. Например, в C++ допустимо следующее:

```
int a = computeSquare(x, y) * 5;          // разрешается в C++
```

Не очень изящное решение разработчиков компилятора. Вот почему старые версии компиляторов C и C++ не поддерживают это средство. Но разве выше не говорилось, что в вычислениях можно использовать возвращаемое функцией значение?

```
a = computeSquare(x,y) * 5                // допустимо в C и C++
```

Нужно понимать разницу. В примере из предыдущего раздела показано присваивание. Оно всегда возможно и в C++, и в C, и в любом другом языке. В примере данного раздела представлен случай несколько иного рода — инициализация. Хотя примеры похожи, смысл совершенно разный. При инициализации выделяется память и устанавливается значение. Присваивание имеет дело с объектом (переменной), для которой память уже выделена. Ей уже назначен адрес и, возможно, размещенное по этому адресу некоторое начальное значение. Записанное по указанному адресу значение заменяется на новое. О разнице уже говорилось в главе 2, а к связанным с нею моментам мы еще вернемся.

Несмотря на прогресс в области разработки компиляторов, компилятор C++ не стал двухпроходным. Все компиляторы C — однопроходные и не могут "заглядывать вперед". Вот почему они не в состоянии использовать еще не определенное значение, даже если оно определяется на следующей строке. Например, такая запись будет ошибочной:

```
int a = b, b(5);                          // ошибка в C++
```

Здесь переменную b нельзя использовать для инициализации переменной a. Обратный порядок допустим. (Обратите внимание, что синтаксис инициализации аналогичен вызову функции. В языке C следующая запись не разрешается, но она допустима в C++):

```
int b(5), a = b;                          // это приемлемо
```

Спецификаторы типов

C++ наследует из языка C методы тонкой настройки диапазонов целых чисел. Это осуществляется с помощью спецификаторов типов. Ключевые слова signed, unsigned, short и long изменяют размер памяти, выделенной для целых чисел, или интерпретацию битовой последовательности.

Спецификатор signed используется по умолчанию — указывать его не нужно. Например, следующее определение переменной cnt имеет в точности тот же смысл, что и предыдущее:

```
signed int cnt;                    // signed по умолчанию
```

Спецификатор unsigned можно использовать для переменных, которые не могут принимать отрицательные значения (индексов, счетчиков, данных по остатку, количеству и пр.). Этот спецификатор не изменяет размера области памяти, выделенной для значения (16 или 32 бита), но изменяет интерпретацию последовательности битов. Допустимым диапазоном для целых без знака (unsigned) будет от –32 768 до +32 767 на 16-разрядной машине (но не от 0 до 65 535) и от 0 до 4 294 967 295 на 32-разрядной машине. В листинге 3.2 представлен предыдущий пример, где вместо целого signed используется unsigned. Результат выполнения данной версии программы показан на рис. 3.2. Как можно видеть, проблема на данном этапе исчезла. Конечно, она проявится на верхнем диапазоне чисел unsigned, но проявится по-другому. При переполнении числа unsigned значение возвращается к 0, а не становится большим отрицательным числом. Неизвестно, что лучше.

```
Целочисленное переполнение в C++:
Увеличение от 32765
1   32766
2   32767
3   32768
4   32767
5   32766
Спасибо, что побеспокоились о границах
диапазона целого
```

Рис. 3.2. *При использовании целых значений unsigned переполнение происходит при больших значениях, чем для простых целых*

Листинг 3.2. Демонстрация типа unsigned int

```
#include <limits>
#include <iostream>
using namespace std;
int main(void)
{
  int unsigned num = INT_MAX - 2;
  int cnt = 0;
  cout << "Целочисленное переполнение в C++:" endl;
  cout << "Увеличение от " << num << endl;
  while (cnt < 5)
  { num = num + 1;
    cnt = cnt + 1;
    cout << cnt << " " << num << endl; }
  cout << "Спасибо, что побеспокоились о границах диапазона целого" << endl;
  return 0;
    }
```

```
Отрицательные значения в переменной unsigned
Обратный отсчет, начиная с +1
1   1
2   0
3   65535
4   65534
5   65533
Спасибо, что побеспокоились о границах
диапазона целого
```

Рис. 3.3. *Переменная unsigned не может содержать отрицательных значений. При дальнейшем уменьшении она принимает большие положительные значения. Предупреждение не выводится*

Применение чисел unsigned integer — неплохая идея (не с точки зрения расширения диапазона значений, а в плане уведомления об этом тех, кто будет заниматься сопровождением программы), особенно, когда значение не может быть отрицательным. Но, если программист, сопровождающий программу, не поймет намерений разработчика и использует переменную unsigned integer для отрицательных значений, результат может быть катастрофичным. В листинге 3.3 показана предыдущая версия программы, где переменная num инициализируется значением 2 и весьма неблагоразумно уменьшается при выполнении цикла (см. рис. 3.3).

Листинг 3.3. Отрицательные значения в переменной типа unsigned

```
#include <iostream>
using namespace std;

int main(void)
{
  int unsigned num = 2;
  int cnt = 0;
  cout << "Отрицательные значения в переменной unsigned" endl;
  cout << "Обратный отсчет, начиная с +1" << endl;
  while (cnt < 5)
  { num = num - 1;
    cnt = cnt + 1;
    cout << cnt << " " << num << endl; }
  cout << "Спасибо, что побеспокоились о границах диапазона целого" << endl;
  return 0;
    }
```

Два квалификатора управляют памятью, перераспределяемой для переменных long и short:

```
int cnt; short int short_cnt; long int long_cnt;
```

Цель здесь не только в том, чтобы обеспечить больший диапазон целых значений, а чтобы сэкономить место там, где можно. Обычно для программистов, применяющих C++, имеет значение производительность. Это касается как скорости выполнения программы, так и занимаемой памяти. Применение чисел signed (без спецификаторов типа) дает самый "быстрый" тип данных, а использование спецификатора long может защитить от переполнения (за счет памяти). Используя целые значения short, программист может избежать напрасной траты памяти. Например, переменная cnt в предыдущем примере изменяется от 0 до 5. Зачем выделять для нее 32 бита на современной машине? Достаточно было бы байта. Если память в дефиците, то указание спецификатора short может стать хорошим вариантом.

Насколько важно использование спецификатора short для экономии памяти и long для расширения диапазона значений? Эти спецификаторы типа усложняют программу. Многие программисты применяют их только в том случае, если им известна проблема переполнения (или нехватки памяти) и ясно, что спецификаторы помогут ее разрешить. В остальных ситуациях большинство программистов предпочитают работать с обычными целыми без спецификаторов и не задумываются о подобных вопросах. Особенно это относится к современным 32-разрядным машинам. Использование 4-х байт для обычных целых в какой-то степени защитит программу от переполнения. Наличие достаточных объемов памяти делает указание спецификаторов short излишним.

Как это часто бывает со средствами C++, унаследованными из C, ситуация не вполне такова, как кажется на первый взгляд. Логично предположить, что для целого short памяти выделяется меньше, чем для просто целого, а для целого long — больше. Однако стандарт C (и C++) требует от разработчиков компилятора, чтобы тип short int был не длиннее просто int, а long int — не короче int. На самом деле все не так запутано. На 16-разрядных машинах для переменных short int и int выделяется одинаковое количество памяти, а для переменных long int — 32 бита. На 32-разрядных машинах все как раз наоборот: для short int отводится 16 битов, а для int и long int — 32.

В C++ есть оператор sizeof, вычисляющий размер данных в байтах. Аргументом может быть имя переменной или имя типа. На любой платформе следующее выражение возвращает значение оператора sizeof:

```
sizeof(short int) <= sizeof(int) <= sizeof(long int)
```

Интересное следствие: short int и long int всегда имеют строго установленный размер, независимо от платформы (32- или 16-разрядной), т. е. в любой архитектуре значение short int занимает 16 бит, а long int — 32. Вот почему программисты, озабоченные вопросами переносимости программного кода, не используют простых целых, а применяют short int для относительно небольших значений и long int для всех других, которым не хватает short int. Обычно это проектировщики встроенных и коммуникационных систем. В таких системах память приходится экономить (ее объем ограничен, а цена значительна), а один и тот же программный код должен работать на нескольких аппаратных платформах.



Советуем В зависимости от аппаратных средств, целочисленные значения могут занимать в памяти 16 бит или 32 бита. Значения short int всегда занимают 16 бит, а long int — 32 бита. Их явное указание устраняет проблемы переносимости.

Можно ли скомбинировать спецификатор unsigned и short или long int, как в следующем примере?

```
insigned short int short_cnt;   long unsigned int long_cnt;
```

Да, можно. (Обратите внимание на порядок спецификаторов — это важно.) Такой вариант можно встретить, например, в драйверах контроллеров жестких дисков, где размер файла или число цилиндров требуют больших неотрицательных целых значений. Тем не менее в большинстве приложений лучше избегать излишней сложности и использовать обычные целые.

И еще одно замечание. В начале данной главы уже упоминалось о старом правиле: когда имя типа опускается, по умолчанию это целое. Данное правило применимо и к описанной ситуации. Если используются типы данных short и long, нет никакой необходимости писать ключевое слово int:

```
int cnt;   short short_cnt;   long long_cnt;              // смысл тот же
```

Целочисленные значения можно представлять как десятичные, восьмеричные и шестнадцатеричные. Например, десятичное 64 представляется как восьмеричное 100 или шестнадцатеричное 40. Чтобы избежать путаницы, целочисленные литералы, начинающиеся с 0, обозначают восьмеричную систему, а с 0x — (или 0X) — шестнадцатеричную. Так 100 записывается как 100 в десятичном представлении, но 0100 в восьмеричной записи означает 64 в десятичной, а 0x100 в шестнадцатеричной — это 256 в десятичной.

Для литеральных значений память распределяется точно так же, как для переменных. Единственная разница в том, что их адресами манипулировать нельзя, следовательно, нельзя изменять хранящиеся там значения. Таким образом, для значения 63 может выделяться 2 байта (литерал short) или 4 байта (литерал long). Чтобы указать разницу, в константах часто применяют спецификаторы short или long. Здесь они записываются так: 63s, 63S, 63l или 63L. Аналогично с беззнаковыми значениями: 63u, 63U, 63UL или 63us. На практике их применяют редко.

Символы

Символьный тип интерпретируется в C++ как еще один вид целого. Он имеет размер 1 байт (8 бит) и может представлять любой символ ASCII: букву, цифру или непечатаемый управляющий символ. Вот несколько примеров определения переменных символьного типа:

```
char c, ch;   char first, last;
```

Спецификаторов short и long для символов не существует, но допускаются спецификаторы signed или unsigned. К сожалению, назначения по умолчанию не стандартизированы. На некоторых машинах символ char означает unsigned char, на других — signed char.

Зачем нужно знать такие подробности? Обычно это не требуется, отсюда и символьный тип не стандартизирован. Но от подобных деталей зависит интерпретация значений char в вычислениях. Например, signed char может содержать библиотечную константу "конца файла" (EOF), значение которой определяется как –1. Тип unsigned char содержит только положительные значения. Таким образом, если попробовать записать – в значение unsigned char, то там окажется 255, а не –1. Поскольку символ char неявно определяется как signed char или unsigned char, возможны проблемы переносимости.

Как любую переменную, char можно инициализировать в определении или присвоить значение позднее. Для инициализации или присваивания разрешается использовать небольшие целые значения. Они будут интерпретированы как коды символов. Символьные литералы заключаются в одиночные кавычки. Это могут быть символы, восьмеричные или шестнадцатеричные значения, либо ESC-последовательности. Важно не путать одиночные и двойные кавычки. Одиночные служат для обозначения символьных литералов, а двойные — для строковых литералов (последовательностей или массивов символов):

```
char c = 'A', ch = 65;              // c и ch содержат 'A'
```

Это пример использования символьного литерала в кавычках и применения десятичного целого числа. Другие представления символов начинаются с ESC-символа '\'. Он не интерпретируются как обычный символ — это сигнал компилятору, что следующие символы должны восприниматься особым образом, например как восьмеричное или шестнадцатеричное значение:

```
c = '\0101'; ch = '\0x41';          // восьмеричное и шестнадцатеричное значение 'A'
```

Здесь кавычки и ESC-символы не являются необходимыми. Можно непосредственно использовать шестнадцатеричные и восьмеричные значения, подобно тому, как записано выше десятичное значение 65 (восьмеричный литерал начинается с 0, а шестнадцатеричный — с 0x или 0X):

```
c = 0101; ch = 0x41;                // восьмеричное и шестнадцатеричное значение 'A'
```

ESC-символы необходимы только в том случае, если значения встраиваются в строку. Кроме того, его применение говорит программисту, сопровождающему программу, что здесь она работает с символами, а не с числами. Наиболее распространенная ESC-последовательность — новая строка, '\n'. Другие стандартные ESC-последовательности включают в себя '\r' (возврат каретки), '\f' (перевод формата или новая страница, '\t' (табуляция), '\v' (вертикальная табуляция) и '\a' (звуковой сигнал).

Одиночные и двойные кавычки имеют в C++ особый смысл, поэтому для представления собственно символа одиночной или двойной кавычки также требуется ESC-символ, например, '\'' или '\"'. Это же относится к символу \. Если нужно вывести его на экран, то потребуется два символа — '\\'.

Есть и еще две ESC-последовательности: '\b' и '\0'. Первая обозначает возврат на позиции, а вторая — числовой ноль. Это не печатаемый символ (печатаемый '0' имеет числовое значение 48), однако он важен. Данный символ компилятор C++ вставляет в конец любой строки. Он используется также для поиска конца строки. Например, литеральная строка "Hello" содержит не пять, а шесть символов. Последний, нулевой символ, вставляет компилятор. Поскольку массивы мы еще не обсуждали, то строки подробнее сейчас рассматривать не будем, а вернемся к этому позднее.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
Одиночная ' и двойная " кавычки
имеют специальный смысл
новая строка: то же, что \
```

Рис. 3.4. *Вывод программы (листинг 3.4), выполняющей арифметические операции как символьные переменные*

C++ интерпретирует символы как малые целые. Следовательно, можно выполнять с ними арифметические операции и сравнения. В листинге 3.4 показан пример такого рода операций. Сначала программа выводит алфавит буквами в верхнем регистре, а потом — в нижнем. Она демонстрирует также применение символа ESC для вывода одиночной кавычки, двойной кавычки и самого символа ESC. Вывод программы демонстрируется на рис. 3.4.

Листинг 3.4. Пример операций с символьными значениями

```
#include <iostream>
using namespace std;
int main(void)
{ char ch; int cnt;
  ch = 65; cnt = 0;                                    // ch содержит 'A'
  while (cnt < 26)
    { coun << ch;
      ch = ch + 1; cnt = cnt + 1; }
  cout << endl;
  ch = 'a' - 1;                                        // ch содержит символ '`'
  while (ch < 'z')
    { ch = ch + 1;                                     // ch содержит 'a', 'b',...,'z'
      cout << ch; }
  cout << endl;
  cout << "Одиночная \' и двойная \" кавычки имеют специальный смысл\n";
                                                       // новая строка: то же, что endl
  cout << как и ESC-символ \\" << endl;
  return 0;
  }
```

Выполнение арифметических операций с символами нельзя считать хорошей практикой. Программиста, занимающегося сопровождением программы, это может запутать. Еще одна проблема в том, что подобные операции будут работать лишь на компьютерах с непрерывным набором символов (например, ASCII). Если машина использует не непрерывный набор символов (например, EBCDIC), то получатся непечатаемые символы. Между тем арифметические операции с символами выполняются очень часто, поскольку позволяют писать эстетически приятные программы.

Для символьных переменных отводится один байт. Это означает, что данный символьный тип может поддерживать набор из 256 знаков (включая управляющие и непечатаемые символы). Для английского языка этого более чем достаточно, но затрудняет введение новых наборов символов ASCII. В наборе символов Unicode сделана попытка стандартизировать подобные усилия, так что английские, французские, русские, китайские и японские символы включены в один 16-битовый набор символов.

C++ поддерживает данные попытки, предлагая расширенный набор символов wchar_t (от wide — широкий). Память для значений такого типа выделяется, исходя из одного из целочисленных типов — int или short int. Для представления литеральных значений широких символов используется префикс L (в верхнем регистре):

```
wchar_t wc = L'a';
```

C++ позволяет программисту использовать любой набор символов. Вероятно, для переносимости ASCII будет лучшим вариантом.

Булевы значения

Большинство современных языков программирования поддерживают булевы значения, которые могут быть равны только true или false. Они особенно полезны в логических выражениях и для выбора выполняемых ветвей программы.

Язык C не поддерживает булев тип. В нем просто любое ненулевое значение интерпретируется как true, а нулевое — как false. Он позволяет выполнять с этими значениями логические операции (опять же любой ненулевой результат интерпретируется как true, а нулевой — как false). Для обработки логических выражений этого достаточно, однако могут возникать ошибки, не идентифицируемые компилятором. Первоначально C++ унаследовал тот же подход, однако в новом стандарте делается попытка исправить данную ситуацию. Для этого введен тип bool с двумя значениями — true и false.

```
bool flag = false, result = true;
```

Именно "делается попытка", так как применение булева типа не устраняет способствующих ошибкам средств, которые унаследованы из языка C. В C++ они используются как вполне легальные. Сами булевы значения интерпретируются как короткие целые. Если вывести значения flag и result из приведенного примера, то первое даст 0 (а не true), а второе 1 (а не false).

Булевы значения занимают в памяти только один байт. Поскольку булев тип может содержать лишь два значения, под них можно было бы отвести еще меньше памяти. Достаточно одного бита, т. е. компилятор может упаковать 8 булевых значений в 1 байт, однако это потребовало бы дополнительного кода на упаковку и распаковку значений, так как современные компиляторы могут адресоваться только к байтам, а не к битам, а некоторые — к паре байтов. Что касается скорости выполнения, то нынешние компьютеры обладают наибольшим быстродействием, когда обращаются к единицам памяти по четыре байта. Вот почему для целых чисел на многих современных машинах отводится 4 байта.

Начав работать с операциями отношения и логическими операциями, вы познакомитесь с булевыми значениями подробнее.

Типы с плавающей точкой

Целые и символьные значения — это так называемые интегральные типы данных. Значения этих типов отличаются друг от друга только на величину, кратную 1. Они не могут содержать дробную часть, т. е. хранящаяся в них последовательность битов не может интерпретироваться компилятором C++ как дробь. Если нужна дробная часть, то потребуются другие типы.

В C++ нет чисел с фиксированной точкой, что позволило бы программисту управлять числом цифр после десятичной точки (запятой). Вместо этого он может использовать значения с плавающей точкой, состоящие из мантиссы (с целой и дробной частью) и экспоненты. Экспонента выражается в литеральных числах как степень десяти, но в памяти компьютера, конечно, представляется как степень двойки.

В C++ есть три типа с плавающей точкой: float, double и long double. Такой тип как short double отсутствует. Для этого есть тип float. Размер данных типов зависит от машины. Обычно для типа float выделятся 4 байта, для double — 8 байт, а для long double — 10 байт (или 8, как для double).

Число цифр мантиссы, которая может представляться числом с плавающей точкой, зависит от машины. Обычно значения типа float содержат 7 цифр, значения double — 15 цифр, а long double — 19.

Диапазон значений зависит от числа битов, выделенных для экспоненциальной части. Для значений float — экспонента имеет диапазон от –38 до 38. Для значений double диапазоном будет от –611 до 611. Для long double это от –4932 до 4932 (более чем достаточно для всех данных Вселенной).

Приведем примеры определений переменных с плавающей точкой:

```
float pi;  double r;  long double d;
```

Цель иерархии типов с плавающей точкой та же, что и для целых типов — программист получает возможность выбора между размером памяти, точностью представления и величиной значений, которые можно использовать для переменной. Для приложений, где важна не точность вычислений, а главным образом объем памяти (как в большинстве встроенных систем реального времени), можно использовать тип float. В приложениях, где нужна высокая точность вычислений (например, в навигационных задачах), предпочтительным будет тип long double, хотя он занимает больше места, а вычисления с ним выполняются медленнее. Для прочих задач подойдет тип double.

Наверное, справедливости ради нужно отметить, что тип float слишком короток для большинства приложений, а тип long double — слишком "дорог" (в смысле памяти и времени). Все функции в математических библиотеках C++ ожидают аргументов double и возвращают значения double (как функция pow() в программе C++ главы 2).

Типы данных с плавающей точкой имеют фиксированную точность. Например, очень большие и очень малые числа типа double будут содержать на данной платформе одинаковое число цифр. Как уже упоминалось, в C++ отсутствуют типы данных с фиксированной точкой (с заданным числом цифр после десятичной точки).

Типы с плавающей точкой представляются в позиционной системе счисления (с десятичной точкой) или в экспоненциальном виде (буква E или e обозначает экспоненту).

```
double r=5.3;  long double d=530.0e-2
```

Эти два числа представляют одно и то же значение. 10 в степени –2 есть единица, деленная на 10 в степени +2, т. е. 1/100.

В данном случае экспоненциальное представление не дает преимуществ по сравнению с нормальным (позиционным). Оно удобно для компактного представления очень малых или очень больших значений.

Большинство литералов с плавающей точкой содержат все три компонента мантиссы: целую часть, десятичную точку (одну, конечно) и дробную часть. Не все они необходимы для представления чисел в каждом конкретном случае.

Между тем важно использовать это представление так, чтобы значение с плавающей точкой отличалось от целого. Например, иногда можно опустить в числе целую или дробную часть:

```
double small = .09, large = 5.;
```

Можно даже опустить дробную и целую часть, если от целого число будет отличаться экспонентой:

```
double big = 500e2;
```

В экспоненциальном представлении экспонента должна быть целым числом. Хотя с математической точки зрения это не обязательно, C++ воспринимает только целые экспоненты. Кроме того, экспонента может быть значением signed, даже если оно положительное:

```
double big = 500e+2;                    // big = 500e+2.2; не воспринимается
```

Подобно целым литеральным значениям, C++ позволяет различать литералы разных типов с помощью предшествующих значению спецификаторов. Спецификатор f или F обозначает значение с плавающей точкой (float), а спецификатор l или L — значение long double. Можно предположить, что спецификатор d или D обозначает значение с плавающей точкой с двойной точностью. Логично, но неверно. Литералы с плавающей точкой имеют тип double по умолчанию:

```
float pi = 3.14f; double r = 5.3; long double d = 5.3 L;
```

Работа с выражениями C++

Выражения состоят из операндов и операций. Операнды — это все, что имеет типизированное значение, т. е. переменные, возвращающие значения функции, выражения, состоящие в свою очередь из операндов и операций. Операции — это символы, смысл которых в C++ зарезервирован. Применение операции к операндам дает значение — его можно использовать в другом выражении. Пробелы используются для удобства чтения, но они не обязательны:

```
x = (a + b) * (a + 2*b) * (a+3*b);          // пробел не обязателен
```

На порядок вычислений влияют два атрибута операции: старшинство (операция с более высоким уровнем старшинства выполняется первой) и ассоциирование (операции с одинаковым старшинством выполняются слева направо или справа налево).

В C++ имеется 56 операций с 18 уровнями приоритетов. Список операций приведен в таблице 3.1. Невозможно запомнить эту таблицу, просто прочитав ее. Она приводится только для информации, а не для запоминания. Программисты запоминают операции постепенно, в процессе работы. Если есть сомнения в старшинстве операций, используйте скобки. Кроме того, даже если выучить старшинство операций наизусть, программист, занимающийся сопровождением программы, может не знать данной таблицы и спутать порядок вычисления.

Таблица 3.1

Операции C++

Категория	Операция	Ассоциирование
Область действия	::	Слева направо
Основные	() [] -> typeid dynamic_cast static_cast reinterpret_cast const_cast	Слева направо
Вспомогательные	++ -- ~ ! + – * &	Справа налево
	Sizeof new delete (type)	Справа налево
Выбор компонента	->* .*	Слева направо
Мультипликативные	* / %	Слева направо
Аддитивные	+ –	Слева направо
Сдвиг	<< >>	Слева направо
Отношение	< <= > >=	Слева направо
Сравнение	== !=	Слева направо

Таблица 3.1 (*продолжение*)

Операции C++

Категория	Операция	Ассоциирование
Поразрядное И	&	Слева направо
Исключающее ИЛИ	^	Слева направо
Включающее ИЛИ	|	Слева направо
Логическое И	&&	Слева направо
Логическое ИЛИ	||	Слева направо
Присваивание	= *= /= %= += -= <<= >>= &= != ^=	Справа налево
Условные	?:	Справа налево
Перемещение	throw	
Запятая	,	Слева направо

Высокоприоритетные операции

В начале таблицы следуют операции самого высокого приоритета, выполняемые в первую очередь. Например, высокоприоритетными операциями являются круглые скобки. Независимо от того, какие другие операции задействованы в выражении, сначала вычисляются подвыражения в скобках.

Теперь об операциях '+' и '-'. Это унарные операции, т. е. у них только один операнд. Например, +2.0, –2.0. Чем же отличается унарная операция '+' и '-' от сложения и вычитания — низкоприоритетных операций? Унарная операция применяется только к одному операнду, а сложение и вычитание — к двум. Это отличие позволяет записывать смешанные выражения, не требующие для вычисления дополнительных скобок, например 2,5– –0.25. Поскольку унарный минус представляет операцию, ее можно отделить от операнда любым числом пробелов. Вычислению это не помешает. Конечно, чтобы работа была оценена программистом, который будет программу сопровождать, лучше записать данное выражение так: 2,5– (–0.25).

Операция `sizeof` уже упоминалась выше. Это единственная операция C++, которая может применяться как к переменной, так и к идентификатору типа:

```
int x = sizeof(int);   int y = sizeof(x);          // одни и те же значения
```

Здесь переменной x присваивается число байтов, выделяемых для любого целого, а в y записывается число байтов, выделенных для конкретной переменной x (в данном случае она целого типа). Как любую унарную операцию, `sizeof` можно использовать, не заключая в скобки операнд, если это имя переменной:

```
int x = sizeof(int);   int y = sizeof x;           // те же результаты
```

Если же операндом является имя типа, то скобки потребуются — в этом случае они обязательны:

```
int x = sizeof int ;   int y = sizeof x;.          // непорядок
```

К сожалению, это все, что можно сказать пока о высокоприоритетных операциях. Позднее мы к ним вернемся.

Это же относится к операциям выбора компонента (следующий уровень приоритета) и к операции `throw` в конце таблицы. Оператор приведения типа (`cast`) мы обсудим чуть ниже.

Арифметические операции

Пятый и шестой уровень приоритета в таблице 3.1 имеют мультипликативные и аддитивные операции, но здесь вряд ли стоит их обсуждать. У умножения и деления более высокий приоритет, чем у сложения и вычитания. Если нужно изменить порядок вычисления, используются круглые скобки:

```
x = (a + b) * (a + 2*b) * (a+3*b);                    // будьте внимательны
```

В случае переполнения в C++ не возникает никаких исключительных ситуаций (exception): программист сам должен позаботиться о том, чтобы переполнения не произошло, какие бы данные не вводились и не обрабатывались программой.

Арифметические операции выполняются с целыми значениями или значениями с плавающей точкой. При этом операция '/' будет вести себя по-разному. Если операндами являются значения с плавающей точкой, то в результате также получается значение с плавающей точкой, вычисленное с соответствующей точностью. Для целых результатом будет целое, усекаемое в сторону нуля. Например, 7/3 дает 2.333333 для операндов с плавающей точкой и 2 для целочисленных операндов.

Операция получения остатка целочисленного деления (%) дает остаток. Она допускается только для целочисленных операндов и символов, но не для значений с плавающей точкой. Например, 7/3 дает 2 с остатком 1. Следовательно, 7 % 2 будет 1. Если 9 разделить на 3, будет 3 без остатка. Тогда 9 % 3 равно нулю:

```
int x1=7, x2=8, x3=9;  int r1, r2, r3;
r1 = x1 % 3; r2 = x2 % 3; r3 = x3 % 3;        // r1 равно 1, r2 равно 2, r3 равно 0
```

То же самое происходит, если первый операнд меньше второго. В этом случае получается только остаток — сам результат считается равным нулю. Например, операция остатка целочисленного деления 5 на 7 даст 0 и 5 в остатке. Следовательно, 5 % 7 равно 5. Аналогично, деление 6 на 7 дает 0 и 6 в остатке. Тогда 6 % 7 равно 6. При делении 7 на 7 получается 1 без остатка. Отсюда 7 % 7 равно нулю:

```
int a1=5, a2=6, a3=7; int r1, r2, r3;
r1 = a1 % 7; r2 = a2 % 7; r3 = a3 % 7;                  // r1 равно 5, r2 равно 6,
                                                        // r3 равно нулю
```

Для положительных операндов все это довольно просто. Для отрицательных результаты зависят от машины. К счастью, нет никакой необходимости использовать операцию получения остатка целочисленного деления с отрицательными операндами. Обычно она используется, когда нужно определить, есть ли свободное место в объекте-контейнере или он уже заполнен данными (и нужно перейти к началу контейнера). Длина контейнера и позиция следующего элемента в нем не бывают отрицательными.

Ассоциирование слева направо означает, что когда в выражение включаются несколько операций одного приоритета, они вычисляются слева направо. Для умножения и сложения это не важно, но важно для вычитания и деления. Это все равно, что вычислять a + b + c как (a + b) + c или как a + (b + c). Нужно убедиться, что выражение a - b - c вычисляется как (a - b) - c, а не как a - (b - c). Аналогично, a/b/c означает (a/b)/c но не a/(b/c).

Операции инкремента '++' и декремента '--' являются своего рода торговой маркой программирования на C и C++. Это операции сложения и вычитания с одним операндом, равным 1. Следовательно, нужно определить только один операнд. Они реализуют обработку в стиле ассемблера: единица прибавляется или вычитается как непрерываемая операция высокого приоритета. По существу, данные операции создают для своих операндов побочный эффект:

```
int x = 6, y = 10;  x++; y-;                    // теперь x равно 7, y равно 9
```

В своей базовой форме операции эти очень просты. Операция инкремента увеличивает свой операнд на 1, а операция декремента — уменьшает его на 1. Этот пример в точности эквивалентен следующему:

```
int x = 6, y = 10;  x = x + 1; y = y - 1;          // теперь x равно 7, y равно 9
```

Программисты, знакомые с другими языками, часто удивляются, зачем нужны операции инкремента и декремента, если они эквивалентны обычному сложению и вычитанию. Раньше можно было бы ответить, что для операций декремента и инкремента компилятор генерирует более эффективный объектный код, чем для обычного сложения и вычитания. Теперь при современной технологии разработки компиляторов и оптимизации кода разницы в производительности нет.

Сегодня применение данных операций — в основном дело вкуса и стиля. Конечно, никто не вынуждает использовать операции инкремента и декремента, можно обойтись сложением и вычитанием. Программа будет столь же элегантна, корректна и быстра, как программа с операциями инкремента и декремента. Разве что ваш начальник или коллеги усомнятся, так ли уж хорошо вы знаете C++, как вам кажется.

На самом деле операции инкремента и декремента весьма многосторонние. Они не ограничиваются целочисленными значениями. Можно использовать и значения с плавающей точкой:

```
float small = 0.09;  small++;                      // теперь small равно 1.09
```

Кроме того, существуют две формы операций инкремента и декремента: префиксная и постфиксная. В предыдущих примерах применялась постфиксная форма, в которой операция следует за модифицируемым операндом. В префиксной форме операция предшествует операнду. Вот пример использования префиксных операций:

```
int x = 6, y = 10;  ++x; --y;                      // теперь x равно 7, y равно 9
```

В чем же разница? Похоже, что результат тот же. В действительности, в данном контексте префиксные операции эквивалентны следующему коду:

```
int x = 6, y = 10;  x = x + 1;  y = y - 1;         // теперь x равно 7, y равно 9
```

Это в точности то, что и в примере с префиксной формой. Разница между префиксными и постфиксными операциями — в использовании выражений. Как видно, результатом данных операций будут значения (как и в случае любой операции в C++ — это очень важный принцип). В примерах x++ и ++x возвращает значение 7, а y-- и --y дает значение 9. Эти значения можно использовать в любом другом выражении, где допустимо целое. И вот здесь префиксная и постфиксная операции начинают вести себя по-разному.

Когда используется префиксная операция, значение операнда сначала увеличивается (или уменьшается), а затем полученный результат используется в выражении:

```
int x=6, y=10, a, b;  a = 5 + ++x;  b = 5 + -y;    // a равно 12, b равно 14
```

Обратите внимание на пробелы, предшествующие префиксным операциям. Они полезны, чтобы избежать путаницы. Компилятор (и программист, сопровождающий программу) может не понять запись 5+++x, хотя запись 5+--y будет понятна.

При применении постфиксной операции в выражении сначала используется значение операнда и только затем переменная увеличивается или уменьшается:

```
int x=6, y=10, a, b;  a = 5 + x++;  b = 5 + y-;    // a равно 11, b равно 15
```

Как видно, с помощью операций инкремента и декремента нетрудно написать исходный код, в котором легко запутаться. Возможно, это и так, однако в своих простейших формах эти операции весьма популярны — они часто используются, если на каждой итерации цикла нужно увеличивать или уменьшать значение счетчика или индекса. Посмотрите на листинг 3.1 (или циклы в приводившихся ранее примерах). Ни один опытный программист не стал бы писать их без операций инкремента (см. листинг 3.5: результат, конечно, будет тот же, что и в листинге 3.1).

Листинг 3.5. Демонстрация операции инкремента

```
#include <limits>
#include <iostream>
using namespace std;

int main(void)
{
  int num = INT_MAX - 2;
  int cnt = 0;
  cout << "Целочисленное переполнение в C++:" endl;
  cout << "Увеличение от " << num << endl;
  while (cnt < 5)
  { num++; cnt ++;                                          // операция инкремента
    cout << cnt << " " << num << endl; }
  cout << "Спасибо, что побеспокоились о границах диапазона целого" << endl;
  return 0;
    }
```

Вам скоро понравятся операции инкремента и декремента. Если пока с ними не очень комфортно, нет проблем — применяйте обычные арифметические операции, как в других языках. Между тем, если вы будете вовсе избегать операций инкремента и декремента, то ваш начальник начнет подозревать, что вы не столь уж хорошо владеете C++, как предполагалось. Так что время от времени следует делать так, как делают все.

Если вы читаете эту главу впервые, то следующие два раздела ("Операции сдвига" и "Поразрядные логические операции") можно пропустить — так будет легче воспринимать материал.

Операции сдвига

Далее в таблице операций C++ следуют операции сдвига << и >>. Стоп! Но это же не операции сдвига, а операции извлечения и вставки, которые использовались для вывода в объекте cout и ввода в объекте cin. Да, все верно. Здесь мы имеем дело с техникой разработки под названием *"переопределение операций"* (overloading). Операции сдвига применялись в языке C с незапамятных времен. Разработчики C++ решили применить существующие операции в новом контексте, так что вместо изучения новых операций (или новых ключевых слов) придется изучать новый смысл уже существующих.

Возможно, это и не легче, да и техника переопределения операций не такая уж новая. Например, сколько смыслов имеет обычная операция + в языке C? Она может использоваться как:

1) унарный плюс;

2) для сложения целых чисел;

3) для сложения чисел с плавающей точкой (и эти операции реализованы не так, как операции целочисленного сложения);

4) как часть префиксных и постфиксных операций (мы еще не закончили их обсуждение).

Операции '<<' и '>>' сдвигают биты целочисленного значения влево или вправо. Второй операнд задает число битов, на которые сдвигается первый операнд. На самом деле, все не так плохо, как звучит. Рассмотрим сначала операцию сдвига вправо:

```
int x=5, y=1, result;  result = x >> y;                    // результат равен 2
```

Данная операция сдвигает последовательность битов левого операнда (здесь x, где содержится 5) на число позиций, которые задает правый операнд (в данном случае y со значением 1). Двоичное представление 5 — это 101. Если сдвинуть данную последовательность битов на позицию вправо, получится 10, что соответствует целому значению 2 (или степень двойки, заданная вторым операндом).

Операция << сдвигает биты в обратном направлении. Здесь из 101 получается 1010, что соответствует десятичному 10:

```
int x=5, y=1, result;  result = x << y;                    // результат равен 10
```

Когда биты сдвигаются влево, то первые биты операнда теряются, а правые биты заполняются нулями (как в последнем примере). Аналогично при сдвиге вправо теряются правые биты, но то, что происходит с левыми битами, зависит от машины.

Самым левым битом целого со знаком (signed) будет бит знака. Если он равен нулю, то число положительное, а если 1 — отрицательное. Если число положительное, то проблем нет: нулевой бит знака сдвигается вправо, а нули слева занимают его место. Если же значение отрицательное, то вправо сдвигается единица, и возникает проблема переносимости. На некоторых машинах в бите знака оказывается единица (и распространяется дальше). Это называется *арифметическим сдвигом*. На других в бит знака сдвигается 0 (распространяемый вправо). Такой *сдвиг* называется *логическим*.

Поразрядные логические операции

Эти операции включают в себя поразрядную операцию "И" (&), "исключающее ИЛИ" (^), "включающее ИЛИ" (|) и самую высокоприоритетную операцию поразрядного дополнения — отрицания (~). Первые три операции двоичные, а последняя — унарная (требуется только один операнд).

Логические операции, как и операции сдвига, работают с последовательностями битов. Операция выполняется отдельно с каждой парой битов двух операндов. Выполнение этой операции дает соответствующий бит результата.

Поразрядная операция "И" устанавливает бит результата в 1, если оба соответствующих бита операндов равны 1, и в 0, если хотя бы один из двух участвующих в операции битов равен 0. В следующих примерах рассматриваются только четыре бита операнда и предполагается, что другие четыре бита содержат нули. Чтобы проиллюстрировать операцию "И", предположим, что первый операнд равен 12 (1100 в двоичном представлении), а второй — 10 (1010 в двоичном виде). Сравним каждый бит первого операнда с соответствующим битом второго. Как видно, только старший бит в обоих операндах равен 1. Остальные содержат разные значения (0 или 1). Следовательно, значением результата будет 1000 (десятичное 8): 1100 & 1010 дает 1000.

Поразрядная операция "включающее ИЛИ" устанавливает результат в 1, если один или оба бита операнда равны 1. Если оба бита нулевые, то результатом

будет 0. Для операндов 12 (двоичное 1100) и 10 (двоичное 1010) все биты результата, за исключением самого правого, устанавливаются в 1, что дает двоичное значение 1110 (десятичное 14), т. е. 1100 | 1010 дает 1110.

Поразрядная операция "исключающее ИЛИ" устанавливает результат в 1, если только один из битов операнда равен 1. Если оба бита содержат одинаковое значение (0 или 1), то результатом будет 0. В нашем примере первый и последний биты операндов совпадают, а второй и третий отличаются, что дает двоичный код 0110 (десятичное 6): 1100 ^ 1010 дает 0110.

Операция поразрядного дополнения устанавливает биты результата в обратные значения, т. е. для получения результата она меняет биты операнда на противоположные. Если бит операнда равен 1, то бит результата становится нулевым и наоборот. Например, дополнение 12 (двоичное 1100) дает результат 0011 (десятичное 3), или ~1100 дает 0011.

Эти операции часто применяются, когда в приложении обрабатывается большой объем информации о состоянии. Например, может быть включен или выключен коммуникационный канал, устройство готово или не готово, на проводник подается напряжение или нет, заказчик заслуживает крупной скидки или нет, имеет хороший кредитный рейтинг и т. д. На большой машине можно позволить себе выделить для каждого из этих значений целое, хотя реально используется только один бит (содержащий 0 или 1). На малых компьютерах применяется набор булевых значений, для каждого из которых выделяется байт. Вот почему информация такого рода часто упаковывается в слова, так что каждый бит (*флаг*) в битовой последовательности имеет отдельный смысл. Чтобы выделить из слова состояния значение конкретного бита или установить его, используются операции сдвига и логические операции с константами (конкретными последовательностями битов — *масками*).

Предположим, третий бит справа в слове состояния (пусть оно называется `flags`) означает, что устройство включено. Когда устройство включается, программа должна устанавливать этот бит в 1.

Чтобы установить бит в 1, нужна переменная (назовем ее `onMask`), у которой третий бит имеет значение 1. Если применить к третьему биту переменных `flags` и `onMask` операцию "включающее ИЛИ", то третий бит результата будет устанавливаться независимо от того, каково состояние третьего бита в переменной `flags`. Это быстрее, чем проверять данный бит, а потом выполнять операцию "включающее ИЛИ". Проблема в том, что логические операции нельзя применять к отдельным битам — они применяются одновременно ко всем битам операнда. Это означает, что все биты переменной `onMask` (за исключением третьего бита) должны иметь такое значение, чтобы другие биты переменной `flags` не менялись. Для "включающего ИЛИ" биты должны содержать 0.

Именно так создаются маски для последовательностей битов: отдельные биты устанавливаются в значения, как того требует состояние, а остальные — в значения, не изменяющие существующего состояния. В данном примере в переменной `onMask` третий бит устанавливается в 1, а остальные биты — в 0. Для четырехбитового представления получаем 0100 или десятичное 4:

```
int flags, onMask = 4;
flags = flags | onMask;          // устанавливает третий бит в 1
```

Если устройство выключено, это должно отражаться сбросом третьего бита в 0 с сохранением значений остальных битов. Тогда потребуется другая маска (назовем ее `offMask`), устанавливающая третий бит в 0, и логическая операция "И". Чтобы остальные биты не изменились, нужно установить эти биты маски в 1. Следовательно, переменная `offMask` будет содержать 1011 или десятичное 11.

Впрочем, здесь есть одна тонкость. Данная битовая последовательность равна 11, только когда ее размер 4 бит. Для 8 бит последовательность будет иметь вид 11111011, а десятичным значением будет 244. Для 16 или 32 бит потребуется

еще одно значение. Вот типичный пример проблемы переносимости. Решение здесь достаточно простое. Все эти битовые последовательности представляют отрицание битового набора 0100 для слов разного размера. Следовательно, переносимым методом инициализации переменной offMask будет использование битового шаблона, обратного 0100:

```
int offMask = ~onMask;
flags = flags & offMask;              // при этом третий бит сбрасывается в 0
```

Чтобы проверить, установлен ли третий бит, можно применить операцию "И" к маске onMask и переменной flags. Эта операция устанавливает все биты результата в 0, за исключением третьего бита (поскольку все биты в маске onMask, кроме третьего, содержат 0). Если третий бит переменной flags равен 0 (устройство выключено), то результатом будет 0 (false). Если третий бит переменной flags равен 0 (устройство выключено), то результат отличен от 0 (содержит true). Еще один метод доступа к значениям битов состоит в сдвиге последовательности битов флага на две позиции вправо и применении операции "И" к флагу и маске, содержащей все нули, кроме правого бита. Если результатом будет 1, то бит установлен, а если результат нулевой, то бит также нулевой (об операции проверки на равенство — см. ниже):

```
if (((flags >> 2)&1)==1) cout << "Третий бит установлен\n";        // проверка
```

Те, кто не собирается заниматься разработкой ПО для встроенных и коммуникационных систем, вряд ли станут часто использовать сдвиг значений, а потому им можно не слишком вникать в данный материал. Тем же программистам, которые планируют подобную деятельность, лучше попрактиковаться в применении данных операций. В таких системах они очень распространены.

Операции отношения и равенства

Операции отношения используются во всех приложениях. C++ поддерживает четыре такие операции: меньше (<), меньше или равно (<=), больше (>) и больше или равно (>=). Знаки двухсимвольных операций не должны разделяться пробелами (как и в любых других подобных операциях C++). Эти операции наиболее часто применяются при сравнениях в операторах условия и в циклах. Например, в листинге 3.1 в условии цикла проверяется cnt < 5. Если cnt меньше 5 (при первой итерации), то выполняется тело цикла. Если же значение cnt увеличивается и становится равным 5, то 5 < 5 дает false и цикл завершается. Что может быть проще? Это еще одно наследие языка C, которое не столь просто, как кажется.

C++ не имеет встроенного булева типа со значениями, отличными от целых. Обсуждавшийся выше булев тип реализован как короткое целое: true — это 1, а false — 0. То есть результат сравнения в C++ — не просто true или false (как в других языках программирования), а числа 1 и 0. Размер такого целого равен одному байту, но при необходимости его можно преобразовать в целое большего размера.

Следовательно, x > y принимает значение 1, если x больше y. В противном случае оно равно 0. Выражение x < y равное 1, если x меньше y, иначе это 0. Аналогично, x >= y дает 1, если x не меньше y и 0, если x < y. Значением x <= y будет 1, если x не меньше y, и 0, если x меньше y.

Сказанное не меняет форму простого сравнения и особенности его работы, однако использование логических значений как целых открывает возможности для злоупотреблений. Например, каково значение x > y > z? В большинстве языков программирования это просто синтаксис и все (хотя есть исключения).

В C++ такое выражение вполне законно. Так как операции отношения ассоциируются слева направо, сначала мы сравниваем x и y. Если x больше y, то

результатом будет 1, которая затем сравнивается с z. Если 1 больше z, то значением выражения станет 1. В противном случае это 0. Если же x не больше z, то результатом будет 0, который сравнивается с z. Если 0 больше z, то значением выражения будет 1, а иначе — 0. Сомнительно, чтобы кто-то стал писать подобное выражение, просчитывая все варианты.

Далее в таблице следуют операции равенства. C++ поддерживает две операции равенства: "равно" (==) и "не равно" (!=). Символы этих операций также неразделимы. Если сравнение истинно, операция возвращает 1, иначе она дает 0.

Таким образом, значением выражения x == y будет 1, если x равно y. В противном случае оно равно 0. Значением x != y будет 1, если x не равно y; иначе 0.

Предположим, что нужно присвоить z значение 10, если x равно y, и 9, если x и y содержат разные значения. Во всех языках программирования (включая C и C++) можно записать некую простую и недвусмысленную конструкцию:

```
if (x == y)          // устанавливает значение z в 10 или в 9
  z = 10;
else
  z = 9;
```

Но в C++ возможен такой вариант:

```
z = 9 + (x == y);
```

Понять, конечно, труднее, но такая конструкция, несомненно, элегантнее и умнее.

Ситуация усугубляется тем, что эквивалентом true может быть любое (отличное от нуля) значение, а не обязательно 1. Соответственно оно может использоваться вместо true. Кроме того, в C++ все возвращает true, включая операцию присваивания. Например, данная операция присваивания устанавливает переменную x в значение y и возвращает это значение для дальнейшего использования в выражениях (если необходимо):

```
x = y;
```

Это означает, что, если вы укажите операцию равенства '==' как '=', пеняйте на себя. Такая операция будет не ошибкой, а вполне допустимым выражением C++.

Предположим, например, что в приведенном выше примере значением x будет 1, а значением y — также 1. Тогда z должно присваиваться 10. А теперь посмотрим, что получится в случае ошибки в первом выражении:

```
if (x = y)
  z = 10;
else
  z = 9;
```

Данный оператор устанавливает значение x в y (что не меняет значения x, так как x и y в данном примере содержат 1), возвращает это значение в операторе if, интерпретирует его как true (поскольку оно ненулевое) и устанавливает z в 10. Все замечательно.

Проведя поверхностное тестирование, можно подумать, что такой код работает вполне корректно, но если не остановиться на этом и протестировать его на другом наборе значений, например x равным 1 и y равным 2, то обнаружится, что значение z все еще равно 10, а не 9 (опять же, присваивание x = y возвращает здесь значение 2, а это есть true).

Теперь предположим, что ошибка сделана во втором выражении:

```
z = 9 + (x = y);
```

Такое присваивание будет возвращать новое значение x, которому присваивается значение y, оно складывается с 9 и используется для установки значения z. В результате z будет равно не 9 или 10, а 11.

Те, кто видят это впервые, могут подумать, что проблема невелика, поскольку разница между операциями '=' и '==' заметна и бросается в глаза — ошибку нетрудно обнаружить. Конечно. Никто и не говорит, что разницы нет. Но в отрасли, где ПО разрабатывается коллективно, "охота" за подобными ошибками может отнимать немало времени, энергии, нервов и выливаться в финансовые расходы. Ведь подобные ошибки нередки.

Таким образом, следует всегда проверять правильность написания операции как в присваиваниях, так и в равенствах. Хочется еще раз подчеркнуть это, чтобы привлечь внимание читателя к данному вопросу.



Осторожно! Ошибочная запись операции равенства '==' как операции присваивания '=' не будет синтаксической ошибкой. Она дает допустимое в C++ выражение, и компилятор молча сгенерирует некорректный код. Всегда проверяйте в условиях такие ошибки.

Логические операции

Следующий набор операций — это логические операции: логическое И (&&), логическое ИЛИ (включающее) (||) и логическое отрицание (!). Подобно поразрядным операциям, "И" и "ИЛИ" представляют собой двоичные операции (они требуют двух операндов). Исключающего ИЛИ среди логических операций нет.

Логические операции очень популярны. Они не менее важны, чем операции отношения. Весьма трудно найти программу, где бы они не использовались. Почему же их запись практически повторяет поразрядные операции? Потому что C++ унаследовал эти операции из языка C, а в C эти операции действительно вспомогательные по отношению к поразрядным операциям.

В отличие от поразрядных операций в логических операциях операнды интерпретируются как одно целое: если значение равно 0, оно рассматривается как `false`, а ненулевое значение считается истинным (`true`).

Логическая операция И (&&) возвращает 1 (размера `bool`), только когда оба операнда ненулевые. В противном случае возвращается 0:

```
if (x < y && y < z) cout << "y между x и z\n"
```

Логическая операция ИЛИ (||) возвращает 1, когда оба операнда отличны от нуля. Когда оба операнда равны 0, возвращается 0:

```
if (x > y || y > 0) cout << "По крайней мере одно положительное\n";
```

Логическая операция отрицания (!) возвращает 0, если операнд ненулевой. Если операнд нулевой, возвращается 1. Применения данной операции всегда можно избежать за счет реорганизации других условий, но иногда проще использовать отрицание. Рассмотрим, например, программу, которая дает скидку пенсионерам (`age >= 65`) с хорошим кредитным рейтингом (`rating == 2`). С помощью отрицания этих условий нетрудно выявить тех, кто подобной скидки не заслуживает. Программист может посчитать, что проще всего записать:

```
if (!(age >= 65 && rating == 2)) cout << "Нет скидок\n";
```

В качестве логических операндов можно использовать целочисленные объекты и объекты с плавающей точкой. Любое ненулевое значение дает `true`, а нулевое — `false`. Заметим, что нет необходимости заключать операнды логических операций в скобки. В то же время, логическое выражение в операторе `if` (и в операторе `while`) должно быть в скобках. Вот почему в последнем примере два набора скобок.

Как и в других языках, логические операции вычисляются слева направо, но, в отличие от прочих языков, операция '&&' имеет более высокое старшинство, чем '||'. Это позволяет писать весьма сложные логические выражения без скобок. Пусть, например, нужно предоставить скидку в 10% пожилым гражданам с кредитным рейтингом 2 и покупающим товар впервые, если сумма заказа не менее $200. Это можно выразить так:

```
if (age>=65 && rating==2 || first_time == true && total_order>200.0)
  discount = 0.1;
```

Такой способ записи сложных выражений не всегда наилучший. В сложных выражениях иногда предпочтительнее использовать скобки — тогда занимающийся сопровождением программист будет знать, каковы компоненты выражения:

```
if ((age>=65 && rating==2) || (first_time == true && total_order>200.0)
  discount = 0.1;
```

В данном примере скобки вокруг логических подвыражений не обязательны, но иногда они требуются. Например, пожилым гражданам может предоставляться скидка, если кредитный рейтинг равен 1 или 2. Логическое выражение без скобок будет некорректным:

```
if (age>=65 && rating==1 || rating==2) discount = 0.1;
                                    // некорректное логическое выражение
```

В данном операторе скидка предоставляется старшим гражданам с рейтингом 1 и 2, но не просто пожилым (как уже говорилось, операция "И" имеет более высокий приоритет, чем "ИЛИ"). Применение скобок устранит проблему:

```
if (age>=65 && (rating==1 || rating==2)) discount = 0.1;
                                    // корректное логическое выражение
```



Внимание Логическая операция "И" (&&) имеет более высокий приоритет, чем логическое "ИЛИ" (||). Использование скобок поможет программисту, занимающемуся сопровождением программы, понять смысл сложных выражений.

Логические операции в C++ вычисляются по короткой схеме. Это означает, что сначала вычисляется первый операнд, и, если результат определяется первым вычислением, второй операнд вычисляться не будет. В следующем примере, если x не меньше y, то не имеет смысла проверять, что y меньше x: нельзя будет заключить, что y лежит между x и z, в таком случае второе условие не вычисляется:

```
if (x < y && y < z) cout << "y находится между x и z\n"
```

Операции присваивания

Операция присваивания (и ее варианты) имеет низкий приоритет. Это правильно, поскольку выполнять ее нужно после всех других операций в выражении. Фактически операция присваивания открывает некоторые впечатляющие синтаксические возможности, но в то же время создает определенную опасность. Все, что имеет адрес в памяти, можно использовать в качестве цели присваивания, а его значение разрешается применять непосредственно в других выражениях.

Когда нечто имеет адрес в памяти, говорят об *l-значении* (lvalue — выражении, которое может находиться в левой части оператора присваивания, семантически представляющем собой адрес размещения переменной, массива, элемента

структуры и т. п. в памяти). Это означает просто, что выражение может использоваться слева от присваивания. Оно имеет адрес, и значение данного адреса модифицируется при использовании в качестве цели присваивания. До сих пор нам встречался только один вид l-значения — имя переменной. В C++ существуют и другие l-значения, мы познакомимся с ними позднее. Заметим, что ничто не мешает использовать l-значение справа от присваивания.

Другой вид значения в C++ называют *r-значением* (rvalue — значением в правой части оператора присваивания). Оно имеет значение, но не адрес в памяти, который программа может использовать для изменения данного значения. Примеры r-значений — это литеральные значения, значения, возвращаемые функциями, результаты двоичных операций. R-значения можно использовать только в правой части выражения, но не как цель присваивания. Вот несколько примеров некорректного применения r-значений как l-значений (все они будут помечены как синтаксические ошибки):

```
5 = foo();          // литерал не может использоваться как lvalue
foo() = 5;          // возвращаемое значение не должно использоваться как lvalue
score * 2 = 5;      // результат операции не должен использоваться как lvalue
```

В отличие от других языков, присваивание C++ представляет собой двоичную операцию, которую можно применять как r-значение, т. е. допускается присваивание по цепочке:

```
int x, y, z;   x = y = z = 0;
```

Присваивание ассоциируется справа налево: выражение x = y = z = 0; означает x = (y = (z = 0));, но не (((x = y) = z) = 0);, так как x = y — не l-значение и присваивать ему ничего нельзя. Данное средство легко использовать неправильно.

```
x = (a = b*c)*4;          // допустимо в C/C++
x = a = b*c*4;            // это имеет другой смысл
x = 4*a = b*c;            // синтаксическая ошибка: для 4*a нет l-значения
```

Кроме традиционной операции присваивания, в C++ имеется ряд вариантов — арифметические операции присваивания. Их назначение — сократить арифметические выражения. Например, вместо записи x = x + y; можно записать x += y;. Результат будет тем же. Такие операции присваивания доступны для всех двоичных операций (+=, -=, *=, /=, %=, &=, |= ^=, <<=, >>=). Они почти столь же популярны, как операции инкремента/декремента и используются для тех же целей. Вот пример сегмента кода, в котором вычисляется сумма квадратов первых 100 целых чисел:

```
double sum = 0.0; int i = 0;
while (i++ < 100)
  sum += i*i;                          // арифметическое присваивание
cout << "Сумма квадратов первых 100 чисел равна " << sum << endl;
```

Вот тот же фрагмент исходного кода, в котором используются более традиционные операции:

```
double sum = 0.0; int i = 0;
while (i < 100)
  { i = i + 1;
    sum = sum + i*i; }
cout << "Сумма квадратов первых 100 чисел равна " << sum << endl;
```

Как уже упоминалось ранее, генерируемый компилятором объектный код в обоих случаях будет одинаковым. Разница чисто эстетическая, и каждому программисту, работающему с C++, предстоит оценить выразительность краткой формы операторов.

Условная операция

Следующая в иерархии операций C++ — условная операция. Это единственная тернарная операция в C++. У нее три операнда. Сама операция состоит из двух символов — '?' и ':', но в отличие от других двухсимвольных операций эти символы разделяются вторым операндом. Вот обобщенная форма условной операции:

```
операнд1 ? операнд2 : операнд3          // если операнд1 равен true,
                                         // вычисляется операнд2
```

Здесь операнд1 — тестовое выражение. Это может быть любой скалярный тип (простой, без программно-адресуемых компонентов), включая float. Данный операнд всегда вычисляется первым. Если результат вычисления первого операнда дает true (не ноль), то вычисляется операнд2, а операнд3 пропускается. Если результатом вычисления первого операнда будет false (0), то операнд2 пропускается, и вычисляется операнд3.

Не заблуждайтесь насчет смысла true и false в данном описании. Выражение операнд1, конечно, может быть булевым, но вовсе не обязательно. C++ позволяет использовать любой тип, позволяющий получать 0 и ненулевые значения.

В следующем примере переменной a присваивается самое меньшее из значений y или z. Первым операндом здесь является выражение y < z;. Если выражение равно true, вычисляется операнд2 (здесь — переменная y), и его значение возвращается как значение выражения. Если выражение y < z; отлично от true, то возвращается значение операнда3 (здесь — переменная z):

```
a = y < z ? y : z;          // a устанавливается в минимальное из значений y, z
```

Обратите внимание, что в отличие от применения логических выражений, операнд1 не обязательно заключать в скобки (возможно, применение скобок облегчит чтение). Условная операция разумна и элегантна, но читать ее бывает непросто, особенно когда результат используется в других выражениях. Вот как можно было бы добиться той же цели с помощью оператора if:

```
if (y < z)
  a = y;                    // a устанавливается в минимальное из значений y, z
else
  a = z;
```

Вот еще один пример, демонстрирующий преимущества условной операции. Возвращаемое ею значение используется в другом выражении (оператор вывода). Если сумма баллов у претендента больше 80, то оператор выводит: "Вы приняты"; иначе выводится "Вы не приняты":

```
cout << "Вы" << (score > 80 ? "" : " не")
     << " приняты.\n";
```

Традиционный подход дает более объемный, но простой в восприятии код:

```
if (score > 80)
  cout << "Вы приняты.\n";
else
  cout << "Вы не приняты.\n";
```

Операция запятой

В других языках запятая не интерпретируется как операция. Но не в C++. Она соединяет операнды, вычисляемые слева направо, и возвращает самое правое выражение для дальнейшего использования. Это удобно, если нужно вычислить несколько выражений там, где C++ допускает только одно выражение:

```
выражение1, выражение2, выражение3, ... , выражениеN
```

Здесь вычисляется каждое выражение, начиная с самого левого, так как запятая имеет низший приоритет. Возвращается значение последнего выражения. Это часто используется как побочный эффект самых левых выражений. Вот предыдущий пример, где можно избавиться от ограничителей блока:

```
double sum = 0.0; int i = 0;
while (i < 100)
  i = i+1, sum = sum + i*i;                    // ограничители блока не нужны
cout << "Сумма квадратов первых 100 чисел равна " << sum << endl;
```

Не очень хорошая идея, но интерпретация запятой как операции вполне допустима. Еще один пример намеренного злоупотребления, хотя и относительно безвредный. Более опасно применение запятой как операции, когда это делается ненамеренно. В результате получается некорректный код, в котором не отмечается синтаксическая ошибка, поскольку никаких синтаксических правил C++ не нарушается. Рассмотрим пример с циклом, в котором вычисляется сумма квадратов:

```
double sum = 0.0; int i = 0;
while (i++ < 100)
  sum += i*i,                                  // арифметическое присваивание
cout << "Сумма квадратов первых 100 чисел равна " << sum << endl;
```

Единственная разница между первой версией и этой состоит в том, что в конце тела цикла вместо точки с запятой поставлена запятая. К сожалению, эта ошибка не делает код синтаксически некорректным. Он будет компилироваться и работать. Конечно, неправильно. Результат выведется 100 раз, а не один. Данную ошибку легко найти, но если оператор после цикла был бы не такой простой, то ошибка могла бы стать трудно обнаруживаемой. Нужно знать о том, что операция запятой в неподходящем месте часто маскируется под вполне законную операцию C++.



Осторожно! Ошибочное использование запятой может не дать сообщения компилятора об ошибке, так как запятая — допустимая операция C++.

Смешанные выражения: скрытая опасность

C++ — язык со строгим контролем типов. Это означает, что если контекст требует значения одного типа, то применение значения другого типа даст синтаксическую ошибку. Такой важный принцип позволяет программистам вылавливать ошибки, затрачивая значительно меньше усилий. Вместо тестирования на этапе выполнения достаточно посмотреть на то, что сообщает компилятор.

Рассмотрим, к примеру, тип `TimeOfDay` из главы 2. Это составной тип (не скалярный) с двумя целочисленными компонентами. Есть определенная система записи для установки значений полей и доступа к ним. Это все. Нельзя добавить 2 к переменной `TimeOfDay` или сравнить ее с другой переменной `TimeOfDay` (по крайней мере, не с помощью тех средств C++, которые до сих пор рассматривались). Вот почему следующий сегмент кода будет синтаксически неверным:

```
TimeOfDay x, y;
x.setTime(20,50);  y.setTime(22,40);          // законная операция
x += 4;                  // синтаксическая ошибка: некорректный тип операнда
if (x < y);              // синтаксическая ошибка: некорректный тип операнда
  x = y - 1;             // синтаксическая ошибка: некорректный тип операнда
```

В том что касается числовых типов, C++ — язык со слабым контролем типов. Если бы значения x и y в предыдущем примере имели тип int, первые три строки были бы синтаксически корректны. Они будут корректны и для любого другого числового типа: unsigned int, short, unsigned short, long, unsigned long, signed char, unsigned char, bool, float, double, long double. Более того, они корректны даже тогда, когда x и y принадлежат к разным числовым типам. На этапе выполнения операция будет работать корректно, несмотря на то, что эти переменные имеют разный размер и их битовые наборы интерпретируются по-разному.

Этим C++ существенно отличается от языков с сильным контролем типов. Например, в C++ вполне допустим (и часто используется) следующий код:

```
double sum;
  ...
sum = 1;                    // нет синтаксической ошибки
```

С точки зрения современных языков со строгим контролем типов это явный пример несостоятельности программиста — ошибка будет помечена на этапе компиляции. В одном месте программы утверждается, что переменная sum имеет тип double, а в другом — интерпретируется как целая переменная. Если выводится синтаксическая ошибка, программисту представляется возможность подумать и решить, как устранить несоответствие. Либо нужно определить переменную как целую, либо заменить последнюю строку на

```
sum = 1.0;
```

В современных языках со строгим контролем типов арифметические операции должны выполняться с операндами того же типа. Для программиста, использующего C++, это будет не столь очевидно: приемлемы обе версии оператора, и обсуждать тут особенно нечего.

Конечно, в идеале все операнды выражения должны иметь один тип (согласно принципу строгого контроля типов). Но для целочисленных типов данное правило ослаблено. C++ позволяет смешивать в одном выражении значения разных типов.

На уровне объектного кода C++ следует тем же правилам, что и другие современные языки: все двоичные операции выполняются с операндами одного типа. Комбинироваться разные типы могут только на уровне исходного кода. При вычислении выражений значения числовых типов могут преобразовываться в значения других типов. В результате операции фактически выполняются со значениями одного типа.

Это делается для удобства программиста, чтобы можно было писать смешанные выражения, не выходя за рамки корректного синтаксиса. Но за удобство нужно платить. Платой будет изучение правил преобразования типов и контроль корректности результатов этих преобразований.

В смешанном выражении возможно три вида преобразования типа:

- Приведение размера
- Неявное преобразование
- Явное преобразование (приведение типа)

Приведение размера (расширение) применяется к "коротким" целым типам для преобразования их к "естественному" размеру целого. Такой экзекуции подвергаются значения типа bool, short int, signed char. После извлечения из памяти для использования в выражениях они всегда приводятся к int. При этом значение всегда сохраняется, ведь размера int достаточно для представления любого значения этих более "компактных" типов. В следующем примере складываются два значения short, а результат и его размер выводятся на экран:

```
short int x = 1, y = 3;
cout << "Сумма равна " << x + y <<", а ее размер равен
    " << sizeof(x+y)<<endl;                          // выводит 4 и 4
```

Вычисления выполняются не со значениями типа short, а с соответствующими целочисленными значениями. Преобразование достаточно простое. На 16-разрядной машине оно тривиально, поскольку типы short и int имеют один размер. На 32-разрядной машине к значению short добавляется еще два байта, заполняемых значением бита знака (0 для положительного числа, 1 для отрицательного). При таком приведении размера значение сохраняется.

Аналогично unsigned char и unsigned short int приводятся к int. На 32-разрядной машине это не вызывает проблемы, так как диапазон целых здесь больше, чем диапазон значений short, даже unsigned. На 16-разрядных машинах ситуация иная. Максимальное значение unsigned short здесь равно 65 535, что больше максимального значения int (32 767). Но причин для беспокойства нет. Если значение не помещается в диапазон целого, то компилятор расширяет его до unsigned int. Опять же приведение размера происходит прозрачно для программиста.

Для типов с плавающей точкой все происходит подобным же образом. Значения float приводятся к размеру double. Со значениями float вычисления не выполняются. Когда такое значение считывается из памяти, оно преобразуется к double.

Приведение размеров целых типов и типов с плавающей точкой — нудная техническая задача, но нужно знать о ней, поскольку такие преобразования влияют на время выполнения и могут сказываться в критичных по времени приложениях. Например, при обработке большого количества символов в коммуникационном приложении программист может хранить символы как целые, что позволит избежать приведения их размера при каждом извлечении символа из памяти.

Это типичный для программирования случай выбора между временем выполнения и экономией памяти. Хорошая новость в том, что приведение размера целого не нанесет ущерба с точки зрения корректности программы, чего нельзя сказать о других преобразованиях.

Компилятор выполняет неявные преобразования:

- В выражениях с операндами разных типов
- В присваиваниях (согласно типу целевой переменной)

Когда выражения содержат операнды числовых типов разного размера, к "коротким" операндам применяется "расширяющее" преобразование размера — они приводятся к типу значений большего размера, после чего операция выполняется над операндами одного, большого размера. Если выражение состоит из нескольких операций, то оно вычисляется по правилам ассоциирования операций (обычно слева направо), и на каждом шаге, если это необходимо, преобразуются размеры. Вот иерархия размеров для преобразования в выражениях:

```
int -> unsigned int -> long -> unsigned long -> float -> double -> long double
```

При таком неявном преобразовании сохраняется значение преобразуемого операнда, однако программист должен убедиться, что необходимое преобразование имеет место. В противном случае возможна потеря точности результата (см. листинги 3.6 и 3.7).

Преобразования присваивания изменяют тип правой части присваивания — она преобразуется к типу данных цели присваивания (левой части). Сама операция присваивания всегда выполняется с операндами одного типа. В случае усечения возможна потеря точности, но это не синтаксическая ошибка. Многие компиляторы по доброте дают предупреждение о потере точности, но в C++ такая операция вполне законна. Если программист этого хочет, то он это и получит. Другими словами, работающий с C++ программист имеет все права на подобные действия.

Кроме потери точности, неявное преобразование может давать еще два осложнения: влиять на скорость выполнения и корректность результатов.

Рассмотрим исходный код в листинге 3.6, преобразующий температуру из градусов по Цельсию в градусы по Фаренгейту. Пример вывода этой программы показан на рис. 3.5.

```
Введите, пожалуйста, значения в градусах Цельсия: 20
Значение в градусах по Фаренгейту: 68
```

Рис. 3.5. *Программа из листинга 3.6 дает корректные результаты при неявном преобразовании к* `double`

Листинг 3.6. Демонстрация неявного преобразования типа

```
#include <iostream>
using namespace std;
int main()
{
  float fahr, celsius;
  cout << " Введите, пожалуйста, значения в градусах Цельсия: ";
  cin >> celsius;
  fahr = 1.8 * celsius + 32;                                    // преобразование?
  cout << "Значение в градусах по Фаренгейту: " << fahr << endl;
  return 0;
  }
```

Литерал 1.8 имеет тип `double`. Переменная `celsius` типа `float` перед умножением преобразуется к типу `double`. Литерал 32 имеет тип `int` и преобразуется перед сложением к `double`. В итоге складываются операнды одного типа. Результат вычисления будет иметь тип `double`. Так как переменная `fahr` имеет тип `float`, то результат вычислений перед присваиванием снова преобразуется. Конечно, эти преобразования выполняются очень быстро, но если объем вычислений велик и они многократно повторяются, то быстродействие программы может снизиться. Программист, использующий C++, не должен забывать о производительности или, по крайней мере, быть готовым к обсуждению связанных с нею вопросов.

От подобной проблемы можно избавиться с помощью явных суффиксов типов или вычислений с типами `double`.

Вот пример использования явных суффиксов типов:

```
float fahr,,celsius; ...
fahr = 1.8f * celsius + 32f;                    // float приводится к double
```

А вот пример вычислений с типами `double`:

```
float fahr, celsius; ...
fahr = 1.8 * celsius + 32.0;                    // преобразований нет
```

Даже если производительность — не первоочередная проблема (нередко так оно и есть), программу необходимо писать с учетом удобства ее чтения. Для этого нужно помнить о вопросах, связанных с неявными преобразованиями. Например, при стандартном преобразовании значения температуры из градусов Цельсия в шкалу Фаренгейта используется коэффициент 9/5. Здесь он для простоты превращен в 1,8. Обычно предпочтительнее не считать ничего вручную и реализовать программу, как показано в листинге 3.7. Кроме того, это интерактивная программа и в ней тратится время на ожидание ввода от пользователя и отображения результата. В таком случае несколько преобразований ничего не меняют. Тот же результат программы представлен на рис. 3.6.

```
Введите, пожалуйста, значения в градусах Цельсия: 20
Значение в градусах по Фаренгейту: 52
```

Рис. 3.6. *В результате "отложенного" преобразования к* `double` *программа из листинга 3.7 дает некорректные результаты*

Листинг 3.7. Пример потери точности при целочисленных вычислениях

```
#include <iostream>
using namespace std;

int main()
{
  double fahr, celsius;
  cout << " Введите, пожалуйста, значения в градусах Цельсия: ";
  cin >> celsius;
  fahr = 9 / 5 * celsius + 32;                                    // точность?
  cout << "Значение в градусах по Фаренгейту: " << fahr << endl;
  return 0;
  }
```

Причина некорректности вывода в том, что, вопреки ожиданиям, преобразование из целого в double здесь не выполняется. Поскольку двоичные операции ассоциируются слева направо, целое 9 делится на целое 5 и результатом станет единица. Если же записать строку так:

```
fahr = celsius * 9 / 5 + 32;                    // точность?
```

то все будет иначе. Здесь переменная celsius имеет тип double и все вычисления выполняются с типом double.

Как можно видеть, программисту требуется инструментальное средство, обеспечивающее необходимые преобразования. В C++ для этого предлагается приведение типа (cast) — унарная операция высокого приоритета. Она позволяет явно управлять преобразованиями числовых типов. Операция приведения типа состоит из имени типа в скобках перед преобразуемым значением. Например, (double) 9 преобразует целое 9 к double 9.0, а (int) 1.8 преобразует double 1.8 в целое 1. Теперь небольшое пояснение. Все сказанное соответствует тому, как описывают приведение типа программисты. На самом деле 9 ни во что не преобразуется, оно так и остается целым. В программе создается новое значение типа double, численно эквивалентное целому 9.



Внимание Говорят, что приведение типа преобразует значение. На самом деле оно создает значение целевого типа и инициализирует его с помощью числового содержимого операции приведения типа.

Проблемную строку в листинге 3.7 можно переписать, используя явное приведение типа:

```
fahr = (double)9 / (double)5 * celsius + (double)32;
```

На самом деле, чтобы избежать проблемы усечения, можно написать:

```
fahr = (double)9 / 5 * celsius + 32;
```

При этом целое 9 преобразуется в double 9.0 и , следовательно, целое значение 5 будет неявно преобразовано в double 5.0.

Такая форма приведения типа унаследована C++ из языка C. Язык C++ поддерживает еще одну форму приведения типа, аналогичную по синтаксису вызову функции. В ней имя типа указывается без скобок, а операнд заключается в скобки. Если использовать приведение типа C++, то вычисления из листинга 3.7 будут выглядеть так:

```
fahr = double(9) / 5 * celsius + 32;
```

Кроме того, C++ поддерживает еще четыре дополнительных вида приведения типа: dynamic_cast, static_cast, reinterpret_cast и const_cast. О них будет рассказано позднее. Чтобы показать свои намерения сопровождающему ПО программисту, некоторые применяют явные преобразования к типу, который соответствует выражению. Другие считают, что это только отягощает код и затрудняет задачу сопровождения программы. Третьи вовсе не применяют приведения типа, чтобы не набирать лишнего на клавиатуре.

И еще замечание по вычислению выражений. Уже упоминалось, что операнды обрабатываются слева направо. Это может создать впечатление, что компоненты выражения также вычисляются слева направо, что неверно. C++ не берет на себя никаких обязательств относительно порядка вычисления компонентов выражения. Соблюдается лишь порядок выполнения операций в выражении.

Данное обстоятельство иногда ускользает от внимания программистов. Часто это не важно. Например, в выражении, преобразующем температуру по Цельсию в градусы по Фаренгейту, операции выполняются слева направо. Неважно, в каком порядке вычисляются значения 9, 5, celsius и 32. Они не зависят друг от друга. Важно, когда операции с побочными эффектами используются в другом выражении. Каким, например, будет результат в этом случае?

```
int num = 5, total;
total = num + num++;                                    // 10 или 11?
cout << "Сумма равна " << total << endl;
```

Поскольку здесь применяется постфиксная операция, значение num используется в выражении перед инкрементом, поэтому total будет равно 10. Но это предполагает, что компоненты выражения вычисляются слева направо. При вычислении справа налево num++ подсчитывается первым. Тогда значение 5 сохраняется для использования в вычислениях, а num становится равным 6. Затем вычисляется левый операнд num, но его значение уже равно 6, поэтому total становится равным 11, а не 10.

На моей машине результат равен 10. Возможно, на вашей он будет таким же. Это не означает, что C++ делает "незаконной" любую программу, вычисляющую компоненты выражения слева направо. Как преодолеть проблему? Как сделать так, чтобы на всех машинах получалось 10? Вот как:

```
int num = 5, total;
total = num + num;                                  // 10, а не 11 num++;
cout << "Сумма равна " << total << endl;
```

Нужно получить на всех машинах 11? Легко:

```
int num = 5, total;
int old_num = num; num++;
total = num + old_num;                              // 11, а не 10
cout << "Сумма равна " << total << endl;
```

Всегда можно явно указать, что именно требуется. Так и нужно делать.

Итоги

Итак, в C++ достаточно типов и вычислений выражений. Как можно видеть, всегда полезно подумать о диапазоне используемых типов. Это важно как с точки зрения переносимости программы, так и для корректности результатов. Если нет веских причин делать обратное (например, таково желание вашего начальника), то используйте типы int и double, но убедитесь, что они работают корректно.

Эта глава охватывала большой объем тем, для усваивания данного материала потребуется некоторое время. Поэкспериментируйте с примерами, вернитесь к главе 2 — используйте ее как основу для работы с C++. Не торопитесь применять слишком много продвинутых средств.

Попробуйте комбинировать в выражениях разные типы, но не забывайте о преобразованиях, их влиянии на корректность результатов и производительность программы. Умеренно применяйте явное приведение типа, не используйте выражения с побочными эффектами как часть другого выражения. Избегайте излишней сложности. Вы запутаете компилятор, да и программист, сопровождающий программу, тоже в ней не разберется.

Убедитесь, что вы знаете, что делаете.

Глава 4

Управление ходом выполнения программы C++

Темы данной главы

- ✔ Операторы и выражения
- ✔ Условные операторы
- ✔ Итерация
- ✔ Операторы перехода в C++
- ✔ Итоги

В предыдущей главе обсуждались основы программирования на C++: типы данных и операции, позволяющие комбинировать значения различных типов в выражения. В данной главе рассматривается следующий уровень программирования — реализация алгоритмов для принятия решений и выполнения различных сегментов кода в зависимости от внешних обстоятельств.

Правильное использование управляющих конструкций — один из наиболее важных факторов, определяющих качество программного кода. Когда операторы выполняются последовательно, поочередно, программисту, сопровождающему ПО, достаточно легко в нем разобраться. В каждом сегменте программы есть только один набор начальных условий и, следовательно, один результат вычислений. Однако такие последовательные программы примитивны и мало на что способны. В реальной ситуации нужно выполнять те или иные сегменты кода в зависимости от каких-то условий. Чем более гибок язык программирования с точки зрения управляющих структур, тем больше возможностей в руках программиста.

Когда один сегмент программы выполняется после того или иного сегмента, то наборов начальных условий несколько и возможных результатов вычислений тоже несколько. Запоминать все эти альтернативы становится трудно. Программисты делают ошибки при разработке программы, а те, кто занимается ее сопровождением, ошибаются при чтении исходного кода и внесении изменений. Вот почему современные языки программирования ограничивают возможности программиста при передаче управления от одного сегмента программы к другому. Такой подход называется *структурным программированием*. Программист использует лишь небольшой набор строго заданных конструкций (циклов и операторов условия), и каждый сегмент программного кода имеет только один вход (или два) и один выход (или два).

В C++ реализован компромиссный подход. Он предлагает богатый набор конструкций, позволяющий управлять выполнением программы. Эти конструкции обладают достаточной гибкостью и мощностью для реализации сложных вариантов принятия решений в программе. В то же время они достаточно строгие и не способствуют чрезмерной изощренности программы, затрудняющей ее понимание и сопровождение.

Операторы и выражения

В отличие от других языков в C++ разница между выражением и выполняемым оператором мала: любое выражение можно превратить в оператор, добавив точку с запятой. Вот некоторые примеры допустимых выражений и допустимых в C++ выполняемых операторов:

```
x * y              // выражение, которое можно использовать в других выражениях
x * y;             // оператор (достаточно бесполезный)
a = x * y          // выражение, которое можно использовать в других выражениях
a = x * y;         // оператор, полезный и распространенный
x++                // выражение, которое можно использовать в других
                   // выражениях (только аккуратно)
x++;               // оператор, полезный и распространенный
foo()              // вызов функции, возвращающей значение (допустимое выражение)
foo();             // вызов функции, возвращающей неиспользуемое значение
                   // (допустимый оператор)
;                  // пустой оператор, допустимый, но только запутывающий
```

Как и в других языках, в C++ операторы выполняются последовательно, в том порядке, как они записаны в исходном коде. Логически каждый оператор представляет собой целую, непрерываемую единицу выполнения.

Выполняемые операторы группируются в блоки (составные операторы). Блоки должны ограничиваться фигурными скобками. Синтаксически блок операторов интерпретируется как один оператор и может использоваться там, где предполагается один оператор. Каждый оператор — это блок, завершаемый точкой с запятой, но за закрывающей фигурной скобкой блока операторов точка с запятой не ставится.

Объединение операторов в блок дает два важных преимущества. Во-первых, можно использовать блок с несколькими операторами в том месте, где допускается только один оператор. Во-вторых, можно определять в блоке локальные переменные. Имена таких переменных не будут конфликтовать с именами переменных, определенных в другом месте программы. Первое свойство критически важно для написания управляющих операторов. Без этого невозможно было бы создать никакой реальной программы. Второе свойство имеет важное значение при написании функций. Без него также не написать реальной программы. Вот обобщенная синтаксическая форма составного оператора:

```
{ локальные определения и объявления (если они имеются);
  операторы завершаются точкой с запятой; }              // в конце нет
                                                         // точки с запятой
```

Составной оператор можно использовать как тело функции, как вложенный блок внутри функций или как тело управляющего оператора. Если поставить после завершающей фигурной скобки точку с запятой, то в большинстве случаев это не повредит. Появится лишь пустой оператор, не генерирующий объектный код. Но иногда такая точка с запятой может изменить смысл операторов, так что лучше не использовать ее после закрывающей фигурной скобки (нужно запомнить исключения, когда такая точка с запятой необходима, как, например, в определении класса и в некоторых других случаях).

Составные операторы вычисляются как один оператор — после предыдущего оператора и перед следующим. Внутри блока выполнение также происходит последовательно, в лексическом порядке операторов.

C++ предусматривает стандартный набор операторов управления, которые могут менять ход выполнения программы. Они включают в себя:

условные операторы: `if`, `if-else`

циклы: `while`, `do-while`, `for`

переходы: операторы `goto`, `break`, `continue`, `return`

код с несколькими точками входа: оператор `switch` с ветвями `case`

В конструкции с условиями, в зависимости от булева выражения в условии, оператор может выполняться один раз или пропускаться. В цикле оператор выполняется один или несколько раз, в зависимости от булева выражения цикла. Булево выражение — это выражение, возвращающее `true` или `false`. Его часто называют логическим выражением, выражением условия или просто выражением. В C++ любое выражение можно использовать как булево, что значительно увеличивает гибкость операторов условия и циклов. В других языках применение небулева выражения там, где предполагается булево выражение, даст синтаксическую ошибку.

В операторе `switch` в зависимости от результата вычисления целочисленного выражения выбираются соответствующие ветви `case`. Операторы перехода меняют ход выполнения программы без всякого условия. Часто они используются в сочетании с некоторыми другими управляющими конструкциями (условным оператором, оператором цикла или переключателем `switch`).

Таким образом, для всех управляющих конструкций область действия составляет лишь один оператор. Если логика алгоритма требует выполнения нескольких операторов после проверки логического выражения, то можно использовать составной оператор в фигурных скобках. После закрывающей фигурной скобки точки с запятой не требуется, но каждый оператор в блоке (включая последний) должен завершаться точкой с запятой.

В остальной части данной главы рассмотрены отдельные типы операторов управления выполнением программы C++, даны конкретные рекомендации, что делать и что не следует делать при применении этих управляющих операторов в коде C++.

Условные операторы

Условные операторы являются, вероятно, наиболее распространенными управляющими конструкциями в программах C++. На самом деле, можно написать лишь незначительную часть программного кода, не прибегая к условным операторам. Всегда возникает необходимость выполнить то или иное действие в зависимости от какого-то условия.

Существует несколько форм условных операторов, которые можно использовать в программе C++. Сложные виды условных операторов требуют тщательного тестирования, но их применение часто дает возможность сделать программный код более осмысленным и элегантным.

Стандартные формы условных операторов

Условные операторы в C++ в своей наиболее общей форме имеют две ветви: ветвь `true` и ветвь `false`. При выполнении условного оператора отрабатывается только одна ветвь.

Вот несколько примеров условных операторов в контексте с предшествующим и последующим операторами:

```
предыдущий_оператор;
if (выражение)              // ключевого слова then в C++ нет
  оператор_true;            // обратите внимание на точку с запятой перед else
else
  оператор_false;           // обратите внимание на необязательный отступ
следующий_оператор;
```

Ключевые слова `if` и `else` должны записываться в нижнем регистре. Ключевое слово `then` в C++ отсутствует. Выражение должно заключаться в круглые скобки.

После выполнения `предыдущего_оператора` (им может быть все, что угодно, включая одну из управляющих конструкций) вычисляется выражение в скобках. Логически это булево выражение: нужно знать, принимает условие значение `true` или `false`. Если условие дает `true`, то выполняется `оператор_true`, а `оператор_false` пропускается. Если условие равно `false`, то выполняется `оператор_false`, а `оператор_true` пропускается. Поскольку мы изучаем C++, а не Паскаль, Бейсик, Java или PL/I, то выражение в условии не обязано быть булевым. Это может быть любое выражение любой сложности. Вычисляется его значение, и любое ненулевое значение (не обязательно целое) интерпретируется как `true`, а нулевое — `false`.

После выполнения одного из операторов (`оператор_false` или `оператор_true`) независимо от условия выполняется `следующий_оператор`. Опять же этим оператором может быть все, что угодно, включая одну из конструкций управления.

В листинге 4.1 показана программа, запрашивающая у пользователя температуру в градусах Цельсия, а затем сообщающая, действительно ли это значение (выше абсолютного нуля, т. е. –273 градуса по Цельсию).

Листинг 4.1. Оператор условия

```
#include <iostream>
using namespace std;

int main ()
{
  int cels;
  cout << "\nВведите температуру в градусах Цельсия: ";
  cin >> cels;
  cout << "\nВы ввели значение " << cels << endl;
  if (cels < -273)
    cout <<"\nЗначение " <<cels << " недопустимо\n"                    // нет ;
         << "Оно меньше абсолютного нуля\n";                          // один оператор
  else
    cout << cels<<" - допустимая температура\n";
  cout << "Спасибо, что воспользовались программой" <<endl;
  return 0;
}
```

Обратите внимание на использование ESC-последовательности перевода на новую строку в начале и в конце строк в двойных кавычках, выводимых объектом `cout`, а также на применение манипулятора `endl` в конце программы. Если операционная система не использует буферизацию вывода, то между ESC-символом новой строки `'/n'` и манипулятором `endl` разницы нет. При буферизации `endl` посылает вывод в буфер и "сбрасывает" его, когда он заполняется. Иногда это повышает производительность программы, но многие программисты не беспокоятся об этой разнице.

```
Введите температуру в градусах Цельсия: 20

Вы ввели значение 20
20 - допустимая температура
Спасибо, что воспользовались программой
```

Рис. 4.1. *Вывод программы, представленной в листинге 4.1*

Вывод программы представлен на рис. 4.1.

Обратите внимание на отступы в обобщенном примере условного оператора и в листинге 4.1. Обычно ключевые слова if и else выравниваются на тот же уровень, что и предыдущий оператор. Оператор_true и оператор_false часто выравнивают на несколько позиций вправо. Это делает программу более понятной для сопровождающего ее программиста (и для самого разработчика во время отладки). Глубина выравнивания — дело вкуса. По идее, двух пробелов достаточно. При более глубоком отступе уменьшится размер строки, особенно, если используются вложенные управляющие конструкции, либо сами операторы true/false представляют собой условные операторы, циклы или переключатели.

Заметим, что когда вводится недопустимое значение температуры, программа выводит две строки. Для этого выполняются операторы:

```
cout <<"\n3начение " <<cels << " недопустимо\n";          // ; в конце
cout << "Оно меньше абсолютного нуля\n";                  // 2 оператора
```

Если операторы записываются таким образом, то они должны заключаться в фигурные скобки, т. е. оформляться как составной оператор. Причина в том, что когда программный код используется как часть условного оператора, в каждой ветви есть место только для одного оператора, а не для двух.

А в листинге 4.1 используется другая техника. Оператор cout может быть сколь угодно длинным и занимать несколько строк (важно лишь, чтобы разрыв строки попадал между компонентами оператора, а не оказывался в середине лексемы). Это означает, что некорректно разбивать строку посередине. Но если нужно выводить две строки, то с этим все в порядке.

Оператор_false не обязателен. Если некоторые действия должны выполняться только тогда, когда булево выражения равно true, его можно опустить. Вот обобщенная форма условного оператора без оператора_false:

```
предыдущий_оператор;
if (выражение)
  оператор_true;
следующий_оператор;
```

Этот условный оператор не содержит ключевых слов then или else. Листинг 4.2 показывает урезанную версию программы из листинга 4.1. Пользователь получает предупреждение о неверном вводе (т. е. температура ниже абсолютного нуля), но программа выполняет свою задачу. (Для простоты здесь отображается только заключительная фраза.) Результат выполнения показан на рис. 4.2.

```
Введите температуру в градусах Цельсия: 20

Вы ввели значение 20
Спасибо, что воспользовались программой
```

Рис. 4.2. *Вывод программы, представленной в листинге 4.2*

Листинг 4.2. Условный оператор без части else

```
#include <iostream>
using namespace std;
#define ABSOLUTE_ZERO -273
int main ()
{
  int cels;
  cout << "\nВведите температуру в градусах Цельсия: ";
  cin >> cels;
  cout << "\nВы ввели значение " << cels << endl;
  if (cels < ABSOLUTE_ZERO)
    cout <<"\n3начение " <<cels << " недопустимо\n"
         << "Оно меньше абсолютного нуля\n";                  // 1 оператор
  cout << "Спасибо, что воспользовались программой" <<endl;
  return 0;
}
```

Как и в предыдущем листинге, ключевое слово if выравнивается здесь на тот же уровень, что и предшествующий/последующий операторы, а код в операторе_true выравнивается вправо для демонстрации управляющей структуры.

Обратите внимание на использование именованной константы для абсолютного нуля вместо литерального значения, как было в листинге 4.1. Это считается хорошей практикой программирования — рекомендуется применять именованные константы для каждого литерального значения и помещать их определения в одно место программы, что упрощает сопровождение. Программист будет знать, где искать значение, а одно изменение будет действовать во всей программе. Такой подход намного лучше, чем выискивание каждого вхождения литерала в исходном коде и ошибки из-за пропущенных изменений. В данном маленьком примере –273 — лишь небольшое числовое значение, используемое только один раз. Если потребуется изменить данное значение, то это все равно, что изменить его в тексте программы (кроме того, вряд ли часто потребуется менять значение абсолютного нуля при ее сопровождении). Следовательно, в данном случае все равно, константа это или литерал. Тем не менее применение символических констант — хорошая практика.



Внимание Оператор_true и оператор_false в условиях может быть при необходимости составным оператором.

Листинг 4.3 демонстрирует модифицированную программу из листинга 4.1, где в ветви true используются два оператора, как и в ветви false. Обратите внимание на применение ключевого слова const. Как уже упоминалось выше, это более популярная техника программирования на C++, чем использование директивы препроцессора #define. Вывод программы показан на рис. 4.3.

```
Введите температуру в градусах Цельсия: 20

Вы ввели значение 20
20 - допустимая температура
Вы можете продолжить вычисления
Спасибо, что воспользовались программой
```

Рис. 4.3. *Вывод программы, представленной в листинге 4.3*

Листинг 4.3. Условный оператор с составными операторами в ветвях

```
#include <iostream>
using namespace std;
const int ABSOLUTE_ZERO = -273;

int main ()
{  int cels;
   cout << "\nВведите температуру в градусах Цельсия: ";
   cin >> cels;
   cout << "\nВы ввели значение " << cels << endl;
   if (cels < ABSOLUTE_ZERO)
      { cout <<"\nЗначение " <<cels << " недопустимо\n;"
        cout << "Оно меньше абсолютного нуля\n"; }                  // блок
   else
      { cout << cels<<" - допустимая температура\n";               // блок
        cout << "Вы можете продолжать вычисления\n"; }
   cout << "Спасибо, что воспользовались программой" <<endl;
   return 0;
}
```

В составных операторах должны использоваться открывающая и закрывающая фигурные скобки. Операторы в каждом составном операторе выравниваются немного вправо. Это указывает, что они выполняются последовательно, и помогает

понять реализацию кода при сопровождении программы. Некоторые программисты помещают открывающую и закрывающую фигурные скобки на отдельные строки, считая, что это помогает выделить структуру кода, однако при таком методе программа становится длиннее, а потому труднее ухватить общий смысл кода (особенно, если приходится просматривать ее на дисплее, а не на распечатке). Вот почему вертикальные пробелы лучше использовать реже.

Распространенные ошибки в условных операторах

Условные операторы увеличивают сложность программного кода. Ошибки в таких операторах часто трудно отыскать. Если повезет, то такие ошибки сделают код синтаксически некорректным, но чаще они приводят к неправильному выполнению. Поскольку не все части условного оператора выполняются при каждом прогоне программы, выявление подобных ошибок требует дополнительного планирования и тестирования.

Ошибки часто происходят и в том случае, если сопровождающий программу программист неверно понял намерения разработчика. Это часто случается из-за некорректного выравнивания или некорректного использования фигурных скобок, разграничивающих составные операторы.

Пропущенная фигурная скобка — частая ошибка в управляющих структурах. Предположим, что операторы из листинга 4.3 записываются следующим образом:

```
if (cels < ABSOLUTE_ZERO)
  { cout <<"\n3начение " <<cels << " недопустимо\n";
    cout << "Оно меньше абсолютного нуля\n"; }              // блок
else
  { cout << cels<<" - допустимая температура\n";           // нет скобки
    cout << "Вы можете продолжать вычисления\n";
```

Данная версия программы выглядит корректно и корректно компилируется. И выполняется. По крайней мере, если ввести 20, результат будет точно такой же, как на рис. 4.3. Но если ввести –300, вывод будет выглядеть так, как показано на рис. 4.4.

```
Введите температуру в градусах Цельсия: -300

Вы ввели значение -300

Значение -300 недопустимо
Оно ниже абсолютного нуля
Вы можете продолжить вычисления
Спасибо, что воспользовались программой
```

Рис. 4.4. *Вывод модифицированной программы, представленной в листинге 4.3*

Наверное, понятно, что этот вывод некорректен. Причина в том, что выравнивание вправо видно лишь человеку, но не компилятору. Несмотря на то что оно показывает принадлежность обоих операторов cout к ветви else, компилятор воспринимает все по-другому. Без фигурных скобок он рассматривает второй оператор cout как следующий_оператор, не часть оператора_false. По его мнению, запись соответствует следующей:

```
if (cels < ABSOLUTE_ZERO)
  { cout <<"\n3начение " <<cels << " недопустимо\n";
    cout << "Оно меньше абсолютного нуля\n"; }          // блок
else
    cout << cels<<" - допустимая температура\n";       // нет скобки
cout << "Вы можете продолжать вычисления\n";            // следующий_оператор
```

К счастью, аналогичная ошибка в ветви true оператора if дает синтаксическую ошибку:

```
if (cels < ABSOLUTE_ZERO)
  cout <<"\n3начение " <<cels << " недопустимо\n";
  cout << "Оно меньше абсолютного нуля\n";              // это нонсенс
```

```
else
  { cout << cels<<" - допустимая температура\n";      // блок
    cout << "Вы можете продолжать вычисления\n"; }
```

Здесь компилятор заметит пропуск ключевого слова else, поскольку видит он следующий код:

```
if (cels < ABSOLUTE_ZERO)                  // оператор if без else - нормально
  cout <<"\n3начение " <<cels << " недопустимо\n";
cout << "Оно меньше абсолютного нуля\n";              // это нонсенс
else                                                 // этот else не имеет if
  { cout << cels<<" - допустимая температура\n";     // блок
    cout << "Вы можете продолжать вычисления\n"; }
```

Компилятор думает, что первый оператор cout относится к оператору if без else, а это вполне допустимо. Он полагает, что второй оператор cout — следующий_оператор, что также нормально. Затем компилятор находит ключевое слово else и отказывается это понимать.



Советуем Не забывайте постоянно контролировать фигурные скобки. Они — весьма распространенный источник ошибок.

Важно также обращать внимание на использование точки с запятой в конце оператора C++. Как уже говорилось выше, отсутствие точки с запятой после оператора C++ ведет к неприятностям. Начинающих программистов часто пугает данное правило. Некоторых это правило так беспокоит, что они начинают ставить точку с запятой в конце каждой строки, нужна она или нет. Применение лишней точки с запятой в исходном коде ведет к созданию пустого оператора, который, по наблюдениям, в большинстве случаев никакого ущерба не приносит.

Однако лишняя точка с запятой не всегда так безобидна. Предположим, что директива #define из листинга 4.2 записана таким образом:

```
#define ABSOLUTE_ZERO -273;              // некорректное определение
```

Конечно, это некорректно. Здесь не должно быть точки с запятой (но она должна присутствовать после определения константы в листинге 4.3). Между тем компилятор не сообщает, что данная строка содержит ошибку. Вместо этого он говорит о неправильной записи условных операторов. Причина срабатывания такой директивы #define — в подстановке литерала. Каждый раз, когда препроцессор находит идентификатор ABSOLUTE_ZERO, он подставляет его значение в исходный код. Его значение теперь равно "–273;", а не "–273". Для препроцессора это вполне законно, но компилятор получает от препроцессора следующий условный оператор:

```
if (cels < -273;)                        // точка с запятой в выражении - ошибка
  cout <<"\n3начение " <<cels << " недопустимо\n"
       << "Оно меньше абсолютного нуля\n";          // один оператор
```

Точка с запятой в конце выражения превращает его в оператор. Компилятор сообщает, что выражение cels<ABSOLUTE_ZERO содержит лишнюю точку с запятой. Но в листинге 4.2 можно видеть — данное выражение не содержит точку с запятой. Легко подумать, что причина в чем-то другом, и начать менять вокруг данной строки все, что внушает подозрения. И чем дальше, тем хуже. Расстояние между местом ошибки (директива #define) и ее проявлением (оператор условия) затрудняет анализ. Вот еще одна причина в пользу применения ключевого слова const, а не директив #define.

Иногда точку с запятой ошибочно размещают в конце строки в условном операторе. Предположим, что оператор из листинга 4.3 записан так:

```
if (cels < ABSOLUTE_ZERO);                    // ветвь true
  { cout <<"\n3начение " <<cels << " недопустимо\n";
                                              // следующий_оператор
    cout << "Оно меньше абсолютного нуля\n"; }          // блок
else                                          // else оказывается не в том месте
  { cout << cels<<" - допустимая температура\n";
    cout << "Вы можете продолжать вычисления\n"; }
```

Это синтаксическая ошибка. Чаще всего, компилятор не способен указать неверную строку. Вместо этого, он сообщит вам, что ключевое слово else находится не в том месте. Для компилятора дополнительная точка с запятой в конце условного выражения является допустимым предложением. Это пустой оператор, который не создает проблемы в C++. Я хочу, чтобы этот код напомнил о том, как компилятор воспринимает этот оператор, соответствующим комментарием:

```
if (cels < ABSOLUTE_ZERO)
  ;                                           // не несет функциональной нагрузки
  { cout <<"\n3начение " <<cels << " недопустимо\n";
                                              // следующий_оператор
    cout << "Оно меньше абсолютного нуля\n"; }          // блок
else                                          // else оказывается не в том месте
  { cout << cels<<" - допустимая температура\n";
    cout << "Вы можете продолжать вычисления\n"; }
```

Компилятор видит условный оператор без else, далее блок с двумя операторами и одинокое ключевое слово else. Эта строка с else и помечается как ошибочная (а вовсе не та, где действительно содержится ошибка). В случае ошибок такого рода не нужно тратить время, чтобы понять сообщения компилятора или пытаться реорганизовать структуру исходного кода.

Предположим теперь, что после логического выражения в программе, представленной в листинге 4.2, стоит лишняя точка с запятой. Условный оператор из листинга 4.2 будет выглядеть так:

```
if (cels < ABSOLUTE_ZERO);          // эта точка с запятой определенно вредна
  cout <<"\n3начение " <<cels << " недопустимо\n"
  cout << "Оно меньше абсолютного нуля\n";
```

```
Введите температуру в градусах Цельсия: 20

Вы ввели значение 20

Значение 20 недопустимо
Оно ниже абсолютного нуля
Спасибо, что воспользовались программой
```

Рис. 4.5. *Вывод модифицированной программы, представленной в листинге 4.2*

Такой условный оператор не содержит ключевого слова else, однако отсутствие else здесь не проблема. И такой вариант будет компилироваться без ошибок. Компилятор не даст никакого предупреждения, и остается лишь уповать на тестирование и отладку. Если выполнить эту версию программы, введя значение 20, то результаты будут отличаться от рис. 4.2. Вывод модифицированной программы показан на рис. 4.5.

Ошибки такого рода психологически трудно выявить и при отладке. Если программа дает объемный корректный вывод, то одна лишняя строка может легко ускользнуть от внимания программиста. Таким образом, точки с запятой требуют такой же аккуратности и постоянного контроля, что и фигурные скобки.

Применение конструкций управления ставит другой вопрос — тестирования программы. Проблема не нова, однако управляющие операторы требуют дополнительного планирования, квалификации и, конечно, бдительности. При тестировании последовательных программ обычно достаточно одного прогона. Если

программа корректна, то и результаты будут корректными — не надо тратить время, силы и деньги на дополнительное тестирование. Когда программа некорректна, она дает некорректные результаты. Это будет очевидно с самого начала, если, конечно, программист невнимателен или не торопится заняться другими задачами.



Внимание Все примеры предыдущей главы показывали последовательные программы с единственным маршрутом выполнения. Вот почему корректность программы демонстрирует только один экран.

Даже для последовательных программ однократного выполнения не всегда достаточно. Причина в том, что программа может случайно давать корректные результаты. Для иллюстрации рассмотрим пример преобразования значения температуры из градусов Цельсия в градусы Фаренгейта из листинга 3.7. Там использовался некорректный оператор:

```
fahr = 9 / 5 * celsius + 32;                    // точность?
```

При продумывании ввода тестовых данных важно подумать о простоте тех вычислений, которые придется делать вручную. И это вполне разумно, так как алгоритмы бывают настолько сложными, что ручные расчеты очень трудно сделать корректно. В таком случае можно ожидать, что программист протестирует программу из листинга 3.7, введя 0. Результат показан на рис. 4.6. Как можно видеть, все корректно, так что один набор исходных данных нельзя считать достаточным даже для последовательных сегментов кода.

```
Введите температуру в градусах Цельсия: 0
Значение по Фаренгейту: 32
```

Рис. 4.6. *Вывод программы, представленной в листинге 3.7*

Вернемся к примеру из листинга 4.1. Достаточно ли выполнить эту программу, введя значение 20 (как на рис. 4.1)?. Очевидно, нет, так как в программе есть операторы, которые при таком прогоне не выполняются. А что если эти операторы переводят деньги на счет программиста? Или запускают межконтинентальную ракету? Или приводят к аварийному завершению программы? Или просто дают некорректный вывод? Первый принцип тестирования состоит в том, что тестовые данные должны приводить к выполнению каждого оператора программы как минимум один раз (или более, если нужно защититься от ошибок, скрытых за случайно корректными результатами). Следовательно, для программы из листинга 4.1 потребуется второй тестовый прогон (в дополнение к показанному на рис. 4.1).

```
Введите температуру в градусах Цельсия: -300

Вы ввели значение -300

Значение -300 недопустимо
Оно ниже абсолютного нуля
Спасибо, что воспользовались программой
```

Рис. 4.7. *Второй прогон программы из листинга 4.1*

На рис. 4.7 показан второй тестовый прогон программы из листинга 4.1. Как видно, результаты корректны, что подкрепляет уверенность в правильности программы. Вывод подтверждает, что обе ветви условного оператора работают и дают верные результаты.

Достаточно ли такого тестирования? Вероятно, нет. Если значение абсолютного нуля было набрано некорректно (например, –263 вместо –273), то результаты обоих тестов все равно будут корректными. Таким образом, мы приходим ко второму принципу тестирования. Набор тестовых данных должен включать в себя граничные значения выражений в условиях. Это означает, что в качестве исходных данных должно использоваться значение –273. Если абсолютный 0 был задан как –263, то программа некорректно покажет, что температура –273 недопустима. Другими словами, ввод значения –273 показывает ошибку, которая оказывается скрытой при вводе значения 0 (см. рис. 4.2).

Но и это еще не все. Что если абсолютный 0 был задан как –283, а не –273? Ввод –273 такой ошибки не выявит. Условие –273 < –283 даст false, и программа покажет (корректно), что температура корректна. Это ведет к третьему принципу тестирования: границы выражений условий должны проверяться на true и false.

При целочисленных данных это означает использование в качестве тестового ввода значения –274. В случае данных с плавающей точкой нужно выбрать какое-то небольшое приращение, близкое к граничному значению, например, –273,001 (или что-то иное — в зависимости от контекста приложения).

В общем случае, если код содержит условие x < y, то его нужно тестировать дважды: один раз на x = y (результат должен быть равен false), а другой на x = y - 1 для целых или y минус небольшое значение для данных с плавающей точкой (результат должен быть равен true).

Аналогично, если код содержит условие x > y, то его нужно тестировать дважды: один раз на x = y (результат должен быть равен false), а другой на x = y + 1 для целых или y плюс небольшое значение для данных с плавающей точкой (результат должен быть равен true).

К сожалению, и это еще не все. Данные рекомендации не будут работать с условиями, включающими равенство. Если программа содержит условие x <= y, то тестовый случай x = y даст true, а не false, как при x < y. Для проверки результата false программу нужно протестировать на x = y + 1 (или плюс малое значение). Если же она содержит условие x >= y, то тестовый случай x = y должен возвращать true, а не false; вторым тестовым случаем должна быть проверка на x = y - 1 (или y минус малое значение).

Все это весьма осложняет дело. Каждое условие в программе должно тестироваться отдельно (два тестовых случая на каждое условие), а число тестовых случаев может быть очень велико. У некоторых программистов просто не хватает терпения на анализ, проработку, выполнение и проверку многочисленных тестовых случаев. Они ограничиваются визуальной проверкой кода. Это неправильно. Проверка кода — полезное, но ненадежное средство поиска ошибок.

Когда числа проверяются на равенство (или неравенство), ситуация становится еще более сложной. В листинге 4.4 продемонстрирована программа, которая просит пользователя ввести ненулевое целое и проверяет, равно ли данное число 0 (для защиты от деления на 0). Если число ненулевое, то программа вычисляет обратное значение и квадрат исходного числа. Если введен 0, программа просит пользователя следовать инструкциям. Конечно, пример довольно тривиальный, но без лишних сложностей показывает, какие вопросы возникают в подобных случаях.

Листинг 4.4. Проверка значений на неравенство (некорректная версия)

```
#include <iostream>
using namespace std;

int main ()
{
  int num;
  cout << "\nВведите ненулевое целое число: ";
  cin >> num;
  if (num > 0)                                          // должно быть (num != 0)
    { cout <<"\nВы правильно выполнили инструкции";
      cout <<"\nОбратное значение равно " << 1.0/num;
      cout <<"\nКвадрат данного значения равен " << num * num; }
  else
      cout <<"\nВы не следуете инструкциям";
  cout <<"\nСпасибо за использование данной программы" <<endl;
  return 0;
}
```

```
Введите ненулевое целое число: 20

Вы правильно выполнили инструкции
Обратное значение равно 0.05
Квадрат данного значения равен 400
Спасибо, что воспользовались программой
```

Рис. 4.8. *Вывод первого теста для листинга 4.4*

Заметим, что если вместо `1.0/num` записать `1/num`, то результат будет неверным, так как при целочисленном делении он усекается. Рис. 4.8 показывает, что программа проходит тестирование при вводе значения 20 — она считает вывод корректным.

Поскольку одного тестового варианта недостаточно, программа проверяется с тестовыми данными, нарушающими инструкции. Это позволяет убедиться в выполнении ветви `else` оператора `if`. Как видно из рис. 4.9, программа проходит и этот тест — она сообщает, что пользователь не следует инструкциям.

```
Введите ненулевое целое число: 0

Вы не следуете инструкциям
Спасибо, что воспользовались программой
```

Рис. 4.9. *Вывод второго теста для листинга 4.4*

```
Введите ненулевое целое число: -20

Вы не следуете инструкциям
Спасибо, что воспользовались программой
```

Рис. 4.10. *Вывод третьего теста для листинга 4.4*

Стоп, эта программа неверна! При ее вводе была сделана опечатка: вместо `num != 0` набрано `num > 0`. Кстати, неправильный ввод операции отношения — очень распространенная ошибка. При написании программы с большим количеством числовых расчетов программисты иногда забывают о корректной реализации и тестировании поведения программы с отрицательными числами. Для демонстрации такой ошибки можно протестировать программу в третий раз, введя отрицательное число (см. рис. 4.10). Как можно видеть, программа не принимает ввод и жалуется пользователю на ошибку. Корректная версия программы показана в листинге 4.5.

Листинг 4.5. Проверка значений на неравенство (корректная версия)

```
#include <iostream>
using namespace std;

int main ()
{
  int num;
  cout << "\nВведите ненулевое целое число: ";
  cin >> num;
  if (num != 0)                                          // теперь это корректно
    { cout <<"\nВы правильно выполнили инструкции";
      cout <<"\nОбратное значение равно " << 1.0/num;
      cout <<"\nКвадрат данного значения равен " << num * num; }
  else
      cout <<"\nВы не следуете инструкциям";
  cout <<"\nСпасибо за использование данной программы" <<endl;
  return 0;
}
```

Это подводит еще к одному принципу тестирования. Для выражений условия с операциями равенства нужно использовать три теста: для равенства (должно возвращаться `true`) и для неравенства с каждой стороны (данные тесты должны давать `false`). То же самое относится к выражениям условия с операцией неравенства: проверка на равенства должна давать `false`, а проверка на неравенства — `true`.



Советуем Для операторов `if` с операциями отношения используйте близкие к граничным тестовые значения. Применение значений, сильно отличающихся от граничных, может привести к пропуску ошибки.

Таблица 4.1

Тестовые варианты для простых выражений условия

Выражение	Тест	Результат
x < y	x равно y	False
	x равно y – 1	True
x >= y	x равно y	True
	x равно y – 1	False
x > y	x равно y	False
	x равно y + 1	True
x <= y	x равно y	True
	x равно y + 1	False
x == y	x равно y	True
	x равно y + 1	False
	x равно y – 1	False
x != y	x равно y	False
	x равно y + 1	True
	x равно y – 1	True

Перечисленные принципы сведены в таблицу 4.1. Условие здесь представлено как x операция y, где операцией может быть '<', '>', '<=', '>=', '==', '!='. Для каждой операции в этой таблице приведены тестовые варианты и ожидаемое значение выражения в условии. Предполагается, что x и y — целые. Для чисел с плавающей точкой вместо 1 потребуется небольшое приращение. Для равенства или неравенства с нечисловыми данными достаточно двух проверок, а не трех.

Как видно, тестовые варианты для `x < y` и для `x >= y` одинаковы, но результаты противоположны. Тестовые варианты для `x > y` и `x <= y` тоже одинаковы, но результаты дают обратные. Аналогично тестовые варианты для `x == y` и `x != y` одни и те же с обратными результатами. Это значит, что условия `x < y` и `x >= y` являются отрицанием друг друга. Все условия, где `x < y`, можно переписать как `!(x > y)` и наоборот, а условие, в котором используется `x >= y`, переписывается как `!(x < y)`. Там, где одно условие дает `true`, второе — `false` и наоборот.

Условия `x > y` и `x <= y` также являются отрицанием друг друга, как и `x == y` и `x != y`. Там, где одно условие в каждой паре принимает значение `true`, второе будет равно `false`.

Для приобретения навыков программирования важно хорошо ориентироваться в отрицании логических условий. Для условного оператора это означает правильное форматирование кода. Например, если ветвь `true` содержит много сложных операторов, а ветвь `false` — только два или три, то ветвь `false` может потеряться в программе. Некоторые программисты предпочитают помещать более короткую последовательность операторов в ветвь `true` оператора условия. Например, условный оператор в листинге 4.5 можно записать следующим образом:

```
if (num == 0)                                    // отрицание (num != 0)
  cout <<"\nВы не следуете инструкциям";
else
  { cout <<"\nВы правильно выполнили инструкции";
    cout <<"\nОбратное значение равно " << 1.0/num;
    cout <<"\nКвадрат данного значения равен " << num * num; }
```

Как уже отмечалось, важно не ошибиться и не записать оператор '==' как '='. Это распространенный источник трудноуловимых ошибок. Например, данный условный оператор можно легко записать как

```
if (num = 0)                                     // допустимо в C++
  cout <<"\nВы не следуете инструкциям";
else
  { cout <<"\nВы правильно выполнили инструкции";
    cout <<"\nОбратное значение равно " << 1.0/num;
    cout <<"\nКвадрат данного значения равен " << num * num; }
```

Данный программный код не показывает никаких ошибок этапа выполнения. Некоторые компиляторы могут дать предупреждение, но такая запись в C++ вполне законна. Отдельных программистов раздражают перспективы ошибок такого рода и, чтобы избежать их, они ставят в левую часть сравнения литерал, а справа — переменную:

```
if (0 == num)                          // константа не может быть l-значением
  cout <<"\nВы не следуете инструкциям";
else
  { cout <<"\nВы правильно выполнили инструкции";
    cout <<"\nОбратное значение равно " << 1.0/num;
    cout <<"\nКвадрат данного значения равен " << num * num; }
```

Если ошибочно записать 0 = num, то компилятор помечает это как ошибку, поскольку в C++ литеральные значения не имеют адреса, с которым может манипулировать программа (это r-значение), хотя и хранятся в памяти, как все прочие объекты. В языках программирования можно выделить общую тенденцию: выводить как можно больше ошибок из категории ошибок этапа выполнения в категорию ошибок этапа компиляции. Помнится, при работе с языком Фортран на PDP-11 я как-то ошибочно установил значение константы 1 в 2, так что каждый раз, когда в программе встречалась единица, компилятор использовал значение 2. Все циклы просто сходили с ума, а я не понимал, что происходит.

Еще одной распространенной техникой записи логических условий является использование того факта, что в C++ любое значение дает true, а нулевое — false. Например, многие программисты могли бы записать условный оператор из листинга 4.5 следующим образом:

```
if (num)               // популярная идиома в C++, то же, что и if (num != 0)
  { cout <<"\nВы правильно выполнили инструкции";
    cout <<"\nОбратное значение равно " << 1.0/num;
    cout <<"\nКвадрат данного значения равен " << num * num; }
else
    cout <<"\nВы не следуете инструкциям";
```

С такими идиомами C++ нужно освоиться. Они очень популярны. Если используется обратное логическое условие (т. е. num == 0), то многие программисты запишут данный условный оператор так:

```
if (!num)              // популярная идиома в C++, то же, что и if (num == 0)
  cout <<"\nВы не следуете инструкциям";
else
{ cout <<"\nВы правильно выполнили инструкции";
  cout <<"\nОбратное значение равно " << 1.0/num;
  cout <<"\nКвадрат данного значения равен " << num * num; }
```

Обратите внимание, что запись if !(num) ... некорректна: логическое условие должно заключаться в круглые скобки. Данным средством легко злоупотребить и написать программу, которую будет очень трудно понять.

Обсуждавшиеся до сих пор примеры относительно просты. Чтобы реализовать более сложные задачи, можно использовать составные условия и вложенные условные операторы. В составных условиях тестирование должно выполняться не только на true и false составного условия, но и на каждый возможный исход true и false. Рассмотрим следующий условный оператор, где processOrder() — определенная где-либо функция.

```
if (age > 16 && age < 65)
  precessOrder();
```

```
else
  cout << "Заказчик не привилегированный\n";
```

Ветвь true в данном условии можно протестировать только одним путем: установив age > 16 и age < 65 в true. Ветвь false тестируется двумя способами: установкой age < 65 в false (например, когда age = 65) или установкой age > 16 в false (когда age = 15). Что выбрать? Если только первый способ, то нельзя будет выявить ошибку, когда второе условие некорректно устанавливается в true, например при age > 0. Если же выбрать только второй способ, то не обнаружится ошибка некорректной установки в true второго условия, например когда age < 250. Вот почему необходимо использовать оба способа прохода по ветви false условного оператора. Это намного сложнее, чем тестирование простого условного оператора, но метод вполне естественный. Продумывая тестовые варианты для установки отдельных условий в true или false, можно руководствоваться таблицей 4.1.

Заметим, что мы не тестировали третий способ прохода по ветви false, когда и age > 16, и age < 65 имеют значение false. Некоторые программисты оправдывают пропуск такой комбинации, поскольку условия соотносятся друг с другом: их истинное значение зависит от одной и той же переменной age. Содержимое данной переменной влияет на то, равно ли условие true (середина диапазона значений) или false (нижний и верхний диапазоны age). Значение не может одновременно принадлежать к верхнему и нижнему диапазону. Однако для операции И мы тестируем false для этих условий по отдельности. Тестировать их вместе — только попусту тратить время и деньги.

Аналогичные соображения можно применить и к тестированию составных условий ИЛИ. Рассмотрим следующий пример, в котором сравниваются два значения с плавающей точкой:

```
if (amt1 < amt2 - 0.01 || amt1 > amt2 + 0.01)      // разница больше 1 цента?
  cout << "Разные суммы\n";
else
  cout << "Одинаковые суммы\n";
```

Ветвь false этого оператора можно протестировать только одним способом: установив оба условия в false. Ветвь true тестируется двумя способами: установкой первого условия в true или второго условия в true. Что выбрать? Ответ тот же, что и в случае с операцией И: придется тестировать оба. Лишь это даст гарантию адекватной проверки обоих условий в соответствии с рекомендациями таблицы 4.1.

Хорошая новость в том, что не нужно составлять тестовые варианты для третьего способа прохода ветви true, когда значение true имеют оба условия. Эти условия связаны (оба они зависят от значений переменных amt1 и amt2) и не могут одновременно принимать значение true. Такое тестирование будет избыточным, даже если условия не соотносятся.

Таблица 4.2

Тестовые варианты для составных условий

Операция	Первое условие	Второе условие	Результат
И	True	True	True
	True	False	False
	False	True	False
ИЛИ	True	False	True
	False	True	True
	False	False	False

В таблице 4.2 показано, какие тестовые варианты следует включать в проверку составных условий.

Как уже отмечалось, если условия в составных операторах соотносятся друг с другом, это не особенно влияет на стратегию тестирования. Рассмотрим, например, следующий оператор с зависимыми условиями. Здесь функции processPreferredOrder() и processNormalOrder() определяются где-то в другом месте программы и вызываются в различных ветвях условного оператора.

Заказчик становится привилегированным, если сумма предыдущей сделки превышает $1500, а текущая цена покупки достигла $200.

```
if (amount > 200 && previous_total > 1500)
  processPreferredOrder();
else
  processNormalOrder();
```

Для проверки данного кода потребуются три тестовых варианта. Один должен предусматривать проход ветви `true`, когда `amount > 200` и `previous_total > 1500` равны `true` (например, `amount = 200.01` и `previous_total = 1500.01`). Два тестовых варианта необходимы для прохода ветви `false`. В одном тесте должно устанавливаться в `true` условие `amount > 200`, в `false` — условие `previous_total > 1500` (например, `amount > 200.00` и `previous_total > 1500.01`). Другой тест должен устанавливать в `false` условие `amount > 200`, в `true` — условие `previous_total > 1500` (например, `amount > 200.01` и `previous_total > 1500.00`). Согласно таблице 4.1, каждый тест должен предусматривать проверку граничных условий. Так как они независимы, можно установить в `false` и `amount > 200`, и `previous_total < 1500` (`amount = 200.00`, `previous_total > 1500.00`). Однако такая проверка не позволит выявить все ошибки и не даст уверенности в корректности программы.

Аналогично операции И, операцию ИЛИ с независимыми условиями нужно проверять на трех тестовых случаях: когда первое условие равно `true`, а второе — `false`, когда первое равно `false`, а второе — `true` и когда оба равны `false`. Рассмотрим пример, где `displayRelaxationPackage()` и `displayActivePackage()` — функции, определенные в другом месте программы:

```
if (age > 65 || previous_history == 1)
  displayRelaxationPackage();
else
  displayActivePackage();
```

Тестовый вариант для данного фрагмента программы должен учитывать три ситуации:

- `age > 65` есть `true`, а `previous_history == 1` дает `false`
- `age > 65` есть `false`, а `previous_history == 1` дает `true`
- и `age > 65`, и `previous_history == 1` равны `false`

В первых двух вариантах проверяется ветвь `true` условного оператора, а последний тестовый вариант покрывает ветвь `false`. Поскольку условия в логической операции независимы, их можно одновременно установить в `true`, однако нет необходимости тестировать случай, когда и `age > 65`, и `previous_history == 1` равны `true`, поскольку эта проверка не выявит ошибок, которые не могут показать три предыдущих теста.



Советуем Для операции `&&` нужно тестировать три случая: первое условие равно `false`, второе равно `false` и оба равны `true`. Для операции `||` надо проверять следующие три случая: первое условие равно `true`, второе равно `true`, оба равны `false`.

Вложенные условные операторы и их оптимизация

Вложенные условные операторы очень популярны. Включение условных операторов в ветви другого условного оператора не очень отличается от применения в ветвях иных операторов. Отступ вправо показывает структуру кода и помогает

понять намерения разработчика. Если нужно включить в ветвь несколько операторов, то используется составной оператор в фигурных скобках. Единственный "подводный камень" в применении вложенных условных операторов состоит в соответствии if и else. Каждый else должен соответствовать ближайшему if.

```
if (условие)
  if (условие1)
    оператор_true1;
  else                                          // относится к if с условием1
    оператор_false1;
else                                            // относится к if с условием
  if (условие2)
    оператор_ true2;
  else                                          // относится к if с условием2
    оператор_false2;
```

Это несложный пример: здесь каждый условный оператор представляет собой полный оператор с ветвями true и false. Ситуация может осложниться, если один из операторов true или false пропущен, что обычно происходит, когда программист видит подобие условий в операторах и пытается оптимизировать исходный код, т. е. сделать его более понятным и выразительным.

Рассмотрим фрагмент системы обработки почтовых заказов, в которой определяется сумма заказа и статус покупателя. Если сумма превышает сумму мелкого заказа (пусть это $20), то плата за обслуживание не взимается. Кроме того, привилегированные заказчики получают скидку (10%) и на экран выводится сэкономленная заказчиком сумма. Для мелких заказов скидка не предусматривается. Обычным (непривилегированным) заказчикам обслуживание обходится в $2 за заказ. Как видно, описание процесса довольно длинное. Часто это бывает из-за того, что оно составляется человеком, а человеческий язык не всегда точен и краток. Если подумать, то некоторая избыточность описания на самом деле полезна, поскольку снижает вероятность непонимания, когда программист пытается интерпретировать и осмыслить текст.

Листинг 4.6. Вложенные операторы условия

```
#include <iostream>
using namespace std;

int main ()
{
  const double DISCOUNT = 0.1, SMALL_ORDER = 20;
  const double SERVICE_CHARGE = 2.0;
  double orderAmt, totalAmt; int preferred;
  cout << "\nВведите, пожалуйста, сумму заказа: ";
  cin >> orderAmt;
  cout >> "Введите 1, если заказчик привилегированный, и 0 в противном случае: ";
  cin = preferred;
  if (orderAmt > SMALL_ORDER)
    if (preferred == 1)
      { cout <<"Полагается скидка " <<orderAmt*DISCOUNT<<endl;
        totalAmt = orderAmt * (1 - DISCOUNT); }
    else
        totalAmt = orderAmt;
  else
    if (preferred == 0)
        totalAmt = orderAmt + SERVICE_CHARGE;
```

```
    else
        totalAmt - orderAmt;
  cout < "Общая сумма: " << totalAmt << endl;
  return 0;
}
```

```
Введите, пожалуйста, сумму заказа: 20
Введите 1, если заказчик привилегированный,
и 0 в противном случае: 1
Общая сумма: 20
```

Рис. 4.11. *Вывод для листинга 4.6 (мелкий заказ, привилегированный покупатель)*

В листинге 4.6 показана возможная интерпретация требований. Несмотря на то что в данном коде имеется три условных оператора, фактически выполняется только две проверки (сумма заказа и статус заказчика). Поскольку эти условия независимы, для каждого потребуются два тестовых варианта (мелкий заказ, крупный заказ, привилегированный заказчик и обычный). Результат выполнения такой программы показан на рис. 4.11–4.14.

Представленная в данном листинге реализация весьма хорошо соответствует требованиям, но избыточность многими программистами воспринимается с неудовольствием. Связанное тестирование разных ветвей (`preferred == 1` и `preferred == 0`) наводит на мысль об оптимизации. Обработка в разных ветвях выполняется аналогичная (`totalAmt = orderAmt`), что еще более убеждает в возможности оптимизировать код. Один из способов оптимизации состоит в том, чтобы начать его с присваивания `totalAmt = orderAmt`, а затем проверить, требуется ли модификация в связи со скидкой для крупных заказов, привилегированностью заказчиков или платой за обслуживание для мелких заказов обычных покупателей.

```
Введите, пожалуйста, сумму заказа: 20.01
Введите 1, если заказчик привилегированный,
и 0 в противном случае: 1
Полагается скидка 2.001
Общая сумма: 18.009
```

Рис. 4.12. *Вывод для листинга 4.6 (крупный заказ, привилегированный покупатель)*

```
Введите, пожалуйста, сумму заказа: 20
Введите 1, если заказчик привилегированный,
и 0 в противном случае: 0
Общая сумма: 22
```

Рис. 4.13. *Вывод для листинга 4.6 (мелкий заказ, обычный покупатель)*

```
Введите, пожалуйста, сумму заказа: 20.01
Введите 1, если заказчик привилегированный,
и 0 в противном случае: 0
Общая сумма: 20.01
```

Рис. 4.14. *Вывод для листинга 4.6 (крупный заказ, обычный покупатель)*

Данный метод нередко позволяет устранить часть `else`. Первое решение в листинге 4.6 можно описать следующим псевдокодом:

```
if (некоторое_условие_содержит_true)
  обработка_первого_варианта;
else
  обработка_второго_варианта;
```

Реализуемое оптимизированное решение начинается с обработки второго варианта с последующей его модификацией или сохранением в прежнем виде. Этот псевдокод может выглядеть так:

```
обработка_второго_варианта;
if (некоторое_условие_true)
  обработка_первого_варианта;
```

Данная оптимизированная реализация показана в листинге 4.7. Результат первых двух тестовых вариантов будет в точности таким же, как на рис. 4.11 и 4.12,

```
Введите, пожалуйста, сумму заказа: 20
Введите 1, если заказчик привилегированный,
и 0 в противном случае: 0
Общая сумма: 20
```

Рис. 4.15. *Вывод для листинга 4.7 (мелкий заказ, обычный покупатель)*

```
Введите, пожалуйста, сумму заказа: 20.01
Введите 1, если заказчик привилегированный,
и 0 в противном случае: 0
Общая сумма: 22.01
```

Рис. 4.16. *Вывод для листинга 4.7 (крупный заказ, обычный покупатель)*

однако тесты на обычных покупателей дают результаты, представленные на рис. 4.15 и 4.16. Они отличаются от результатов, показанных на рис. 4.13 и 4.14. Почему?

Листинг 4.7. Оптимизированный вложенный условный оператор

```
#include <iostream>
using namespace std;

int main ()
{
  const double DISCOUNT = 0.1, SMALL_ORDER = 20;
  const double SERVICE_CHARGE = 2.0;
  double orderAmt, totalAmt; int preferred;
  cout << "\nВведите, пожалуйста, сумму заказа: ";
  cin >> orderAmt;
  cout >> "Введите 1, если заказчик привилегированный, и 0 в противном случае: ";
  cin = preferred;
  totalAmt = orderAmt;                    // второй вариант
  if (orderAmt > SMALL_ORDER)             // изменить totalAmt, если не мелкий заказ
    if (preferred == 1)
      { cout <<"Полагается скидка " <<orderAmt*DISCOUNT<<endl;
        totalAmt = orderAmt * (1 - DISCOUNT); }
  else                                    // это оптический обман
    if (preferred == 0)                   // для мелкого заказа проверить статус покупателя
      totalAmt = orderAmt + SERVICE_CHARGE;
  cout < "Общая сумма: " << totalAmt << endl;
  return 0;
}
```

Данная реализация демонстрирует нам "оптический обман". Отступы должны показывать сопровождающему систему программисту (и тестировщику) намерения разработчика, однако это отличается от понимания программного кода компилятором. Согласно правилу соответствия ключевых слов if и else, компилятор интерпретирует условный оператор так:

```
totalAmt = orderAmt;                    // второй вариант
if (orderAmt > SMALL_ORDER)             // изменить totalAmt, если не мелкий заказ
  if (preferred == 1)
     {  cout <<"Полагается скидка " <<orderAmt*DISCOUNT<<endl;
        totalAmt = orderAmt * (1 - DISCOUNT); }
else
  if (preferred == 0)
    totalAmt = orderAmt + SERVICE_CHARGE;          // не обрабатываются для
                                                   // мелких заказов
```

В данном решении, независимо от вида заказчика, мелкие заказы не обрабатываются. (Правильность результатов для мелких заказов и привилегированных заказчиков случайна.) Для крупных заказов некорректно подсчитывается плата за обслуживание. Несмотря на попытку описания одних и тех же действий, понимание человека и компилятора идут здесь разными путями и не пересекаются.

В данном случае нетрудно добиться общей точки зрения. Достаточно лишь поместить в условном операторе фигурные скобки. Кроме того, составной условный оператор не должен содержать несколько операторов — допускается лишь один. Составным он называется не потому, что состоит из нескольких операторов, а потому, что содержит блок в фигурных скобках. Условный оператор из листинга 4.7 должен выглядеть так:

```
totalAmt = orderAmt;                    // второй вариант
if (orderAmt > SMALL_ORDER)             // изменить totalAmt, если не мелкий заказ
  { if (preferred == 1)
    {  cout <<"Полагается скидка " <<orderAmt*DISCOUNT<<endl;
       totalAmt = orderAmt * (1 - DISCOUNT); } }
else
  if (preferred == 0)          // для мелкого заказа проверить статус покупателя
  { totalAmt = orderAmt + SERVICE_CHARGE; }
```

Многие программисты находят такой стиль эффективным и каждый раз используют в условном операторе (или любой управляющей конструкции) фигурные скобки. Это помогает избежать другой распространенной проблемы. Часто все начинается с одного оператора в ветви условия, а потому фигурные скобки не используются. Затем при правке и изменении программы добавляется еще один оператор (или несколько). При этом забывают поставить фигурные скобки, особенно если изменения вносятся сопровождающим ПО программистом. Заключение каждой ветви условного оператора в фигурные скобки снижает вероятность подобных ошибок. Программисту не нужно будет думать об этом при внесении изменений — очень важное преимущество. Так что канонический вид условного оператора следующий:

```
if (выражение)
  { оператор_true; }                    // для будущего расширения
else
  { оператор_false; }                   // для будущего расширения
```

Другой хороший пример вложенных условных операторов и их оптимизации — проблема високосного года. Обычно високосный год без остатка делится на 4. Вот где полезно использовать оператор получения остатка целочисленного деления. В подобной реализации можно написать:

```
if (year%4 != 0)                    // если год не делится на 4, то он не високосный
  { cout <<"Год " <<year <<" не високосный" <<endl; }
else
  { cout << "Год " <<year << " високосный" <<endl;}
```

Для такого простого алгоритма все удивительно точно. Алгоритм накапливает ошибку в 1 день примерно каждые 130 лет. Вот почему, когда данный алгоритм был заменен на еще более точный, используемый уже около 1700 лет, коррекция календаря составила всего 14 дней.

Таким образом, более точное правило состоит в том, что если год делится на 4, то он високосный, а если делится на 100, то нет. Учитывая это, наша программа может выглядеть так:

```
if (year % 4 != 0)                  // если год не делится на 4, то он не високосный
  { cout <<"Год " <<year <<" не високосный" <<endl; }
```

```
else                          // если он делится на 4 - он високосный
  if (year % 100 == 0)        // если он не делится на 100
    { cout <<"Год " <<year <<" не високосный" <<endl; }
  else
    { cout << "Год " <<year << " високосный" <<endl;}
```

Все это правда, но не вся правда. При таком правиле теряется один день на сотню лет, что не очень много. Более корректное правило состоит в том, что когда год делится на 100, он не високосный, если только не делится на 400. Тогда он снова високосный. В таких случаях часто используется операция И (&&) или вложенное условие. Решение проблемы показано в листинге 4.8. Если год не делится на 4, он не високосный. И все. Если делится на 4 и делится на 100, то он не високосный, если одновременно не делится на 400 — тогда год високосный. Если год делится на 4 и не делится на 100, то он високосный. Системный анализ — штука непростая. Представьте, что занимаетесь этим ежедневно.

Листинг 4.8. Решение проблемы високосного года

```
#include <iostream>
using namespace std;

int main ()
{
  int year;
  cout << "Введите год: ";
  cin >> year;
  if (year % 4 != 0)                              // не делится на 4
    cout <<"Год " <<year <<" не високосный" <<endl; }
  else
    if (year % 100 == 0)
        if (year % 400 == 0)                      // делится на 400 (следовательно, на 100)
          cout << "Год " <<year << " високосный" <<endl;
        else                                      // делится на 4 и на 100, но не на 400
          cout <<"Год " <<year <<" не високосный" <<endl;
    else                                         // делится на 4, но не на 400
      cout << "Год " <<year << " високосный" <<endl;
  return 0;
}
```

Здесь три выражения условия, так что сценарий "худшего случая" должен предусматривать шесть проверок. Однако выражения связаны, и выполнять нужно только четыре ветви, поэтому можно обойтись четырьмя следующими тестовыми вариантами:

- год % 4 != 0 есть true (например, 1999)
- год % 4 != 0 есть false (т. е. год % 4 == 0 есть true), год % 100 == 0 есть true и год % 400 == 0 тоже дает true (например, 2000)
- год % 4 == 0 и год % 100 == 0 равны true, но год % 400 == 0 (например, 1900)
- год % 4 == 0 есть true, но год % 100 == 0 равно false (например, 2004)

```
Введите год: 2000
Год 2000 високосный
```

Рис. 4.17
Вывод для листинга 4.8 (год делится на 4, 100 и 400)

На рис. 4.17 показаны результаты выполнения этого кода для 2000 г. В данной программе есть ряд проблем, которые относятся не к ее корректности, а к эстетике. Здесь три уровня вложенности, и условия явно можно объединить. Тестовый случай, когда `year % 4 == 0` равно `true`, содержит две ветви для високосного года, и их также можно скомбинировать. Как это сделать? Во-первых, поэкспериментируем с отрицанием условия. Это поможет сделать ветви ближе друг к другу, например:

```
if (year % 4 != 0)                                  // не делится на 4
  cout <<"Год " <<year <<" не високосный" <<endl;
else
  if (year % 100 == 0)
    if (year % 400 != 0)                            // делится на 100, но не на 400
      cout << "Год " <<year << " не високосный" <<endl;
    else                                            // делится на 4, на 100, на 400
      cout <<"Год " <<year <<" високосный" <<endl;
  else                                              // делится на 4, но не на 100
    cout << "Год " <<year << " високосный" <<endl;
```

Теперь можно объединить два следующих друг за другом условия с помощью операции И. Последние две ветви для високосного года тоже скомбинируются. В результате получится более разумное решение:

```
if (year % 4 != 0)                                  // не делится на 4, точка
  cout <<"Год " <<year <<" не високосный" <<endl;
else
  if (year % 100 == 0 && year % 400 != 0)           // делится на 100, но не на 400
    cout << "Год " <<year << " не високосный" <<endl;
  else                                              // делится на 4, но не на 100
    cout <<"Год " <<year <<" високосный" <<endl;
```

Красиво, не правда ли? Здесь только два уровня вложенности, что вполне приемлемо. Между тем часть обработки для невисокосного года повторяется, а для программиста, применяющего C++, это не очень хорошо. Год не високосный, когда он не делится на 4 или когда условие `if (year % 100 == 0 && year % 400 != 0)` равно `true`. Похоже, нужно использовать операцию ИЛИ, не так ли? В листинге 4.9 показан не только корректный и эффективный, но элегантный и лаконичный ответ.

Листинг 4.9. Решение проблемы високосного года

```
#include <iostream>
using namespace std;

int main ()
{
  int year;
  cout << "Введите год: ";
  cin >> year;
  if (year % 4 != 0 || year % 100 == 0 && year % 400 != 0)
    cout <<"Год " <<year <<" не високосный" <<endl;
  else
    cout <<"Год " <<year <<" високосный" <<endl;
  return 0;
}
```

Выполнение данной программы с уже обсуждавшимися тестовыми вариантами даст те же результаты, какие представлены на рис. 4.7.

Несомненно, программа в листинге 4.9 лучше, чем в листинге 4.8, но вот стоит ли тратить время на удаление 6 строк исходного кода (и проверку корректности полученного результата) — вопрос спорный.

Иногда уходит несколько часов на то, чтобы оптимизировать сложные условные операторы. Нередко результатами можно гордиться, но стоит ли овчинка выделки, оправдано ли это экономически, ясно не всегда, так что смотрите сами.

Итерация

Условные операторы играют в программировании важную роль. Они применяются практически в каждой программе. Но эти операторы справляются с задачей не в одиночку. В каждой программе возникают ситуации, когда требуется повторить ту или иную последовательность операторов для разных заказчиков, транзакций, онлайновых кредитов и пр. Для решения подобных задач нужна итерация.

Для повторяющихся действий C++ предлагает три оператора итерации: циклы `while`, циклы `do-while` и циклы `for`. Каждый вид циклов в C++ управляет повторным выполнением одного оператора (заканчивающегося точкой с запятой) или составного оператора (блока операторов), заключенного в фигурные скобки (после закрывающей фигурной скобки точка с запятой не ставится). Для управления итерациями во всех типах циклов используются логические выражения, аналогичные выражениям в операторах условия. При вычислении этих логических выражений получается `true` или `false` (не ноль или ноль). Они проверяются при каждом повторении цикла. Когда условие цикла становится равным `false`, итерация завершается. Если условие равно `true`, то повторяется тело цикла (оператор или блок). Поскольку в разных типах циклов проверка условия выполняется по-разному, нельзя говорить о проверке условия "перед каждой итерацией". Лучше говорить: "для каждого повторения цикла". Независимо от конструкции цикла в теле цикла делается что-то, влияющее на его условие. Если этого не происходит, то условие может всегда принимать значение `true` — серьезная угроза для любой программы с циклами.

В цикле `while` условие проверяется перед каждой итерацией и итерации прекращаются, когда оно принимает значение `false`. Выражение цикла проверяется и перед первой итерацией, следовательно, возможна ситуация, когда итерации повторятся 0 раз (условие цикла равно `false` при первом вхождении).

В цикле `do-while` условие проверяется после каждой итерации и цикл прекращается, когда оно становится равным `false`. Поскольку в первый раз условие проверяется только после первой итерации, а не перед ней, цикл всегда выполняется хотя бы один раз.

Цикл `for` предназначен для выполнения заранее заданного числа итераций.

Обычно один и тот же алгоритм можно реализовать с помощью цикла любого формата, так что их выбор — дело вкуса. Иногда один формат подходит лучше, чем другой: используется меньше операторов или они лучше соответствуют решению задачи.

Применение цикла while

Цикл `while` выполняется как один оператор. Разница между другими операторами и циклом `while` в том, что тело цикла может выполняться повторно в зависимости от значения логического условия цикла.

Цикл `while` имеет следующую логическую структуру:

```
предыдущий_оператор;
while (выражение)          // выражение цикла
```

```
  оператор;                    // тело цикла
следующий_оператор;
```

Данная управляющая структура повторяет тело цикла, пока его логическое условие (выражение) принимает значение `true`. В конечном счете (или даже перед первым проходом) условие становится равным `false`. Когда это происходит, оператор цикла пропускается и выполняется следующий оператор. Если цикл управляет выполнением нескольких операторов, используются ограничители блока (фигурные скобки):

```
while (выражение)
  { оператор;                  // обратите внимание на отступ
  ...
    оператор; }                // конец тела цикла
```

Распространенной ошибкой является попытка построить цикл так, что в его теле не изменяется значение выражения условия — оно не принимает значения `false`, получается бесконечный цикл, когда программу приходится прерывать средствами операционной системы.

Обычно при проектировании циклов решается некая задача, которую можно условно назвать обработкой текущих данных — элементы данных обрабатываются повторно. Это означает, что для каждого элемента данных необходимы инициализация, вычисление, обработка (печать, использование в вычислениях, сохранение или что-то иное, в зависимости от алгоритма). Затем его нужно изменить для следующей итерации, после чего вычислить и обработать снова. И так далее, пока не будет обработан последний элемент. Шаги обработки можно свернуть в формат цикла `while`, используя следующий образец:

```
инициализация_текущих_данных
 while (вычисление_текущих_данных)          // точка решения
  { обработка_текущих_данных;               // основная цель данного кода
    изменение_текущих_данных; }             // не забывайте про этот шаг!
```

Рассмотрим пример обработки транзакций. Для простоты предположим, что в программе вводятся и складываются 5 величин (ниже будет более реалистичный пример). Поскольку общее число транзакций известно, можно ввести переменную для подсчета уже обработанных транзакций. Это часть текущих данных. Ее нужно инициализировать значением 0 и увеличивать на 1 после каждой транзакции. Другим элементом текущих данных будет сумма введенных значений. Ее также потребуется инициализировать 0 и суммировать на каждой итерации. Но общую сумму нельзя использовать для проверки завершения цикла. А вот счетчик транзакций можно. Следовательно, компонентами цикла будут:

- `Инициализация_текущих_данных`: счетчик устанавливается в 1, общая сумма — в 0
- `Вычисление_текущих_данных`: проверка, превышает ли значение счетчика транзакций 5
- `Обработка_текущих_данных`: ввод следующего значения транзакции и добавление его к сумме
- `Изменение_текущих_данных`: увеличение счетчика транзакций на 1 и переход к его проверке

Реализация представлена в листинге 4.10. Такая конструкция весьма непрактична. Маловероятно, чтобы приложение писалось для конкретного набора данных.

Листинг 4.10. Цикл `while` с бесконечным числом итераций

```
#include <iostream>
using namespace std;

int main ()
{
  double total, amount; int count;
  total = 0.0; count = 1;                  // инициализация текущих данных
  while (count <= 5)                       // оценка текущих данных
    {    cout << "Введите количество: ";
         cin >> amount;                    // ввод текущих данных
         total += amount; }                // обработка текущих данных
  cout << "\nСумма по 5 транзакциям равна "<<total << endl;
  return 0;
}
```

Это типичный пример ошибки программирования. Цикл выполняется, пока значение `count` не превысит 5. Когда `count` станет равно 6, цикл должен завершиться. Проблема в том, что `count` никогда не станет равно 6 (или другому значению), так как в теле цикла `count` не изменяется. Для исправления ситуации нужно на каждой итерации в теле цикла увеличивать `count` на 1. Можно делать это, например, в начале цикла:

```
while (count <= 5)              // вычисление текущих данных
  { cout << "Введите количество: ";
    cin >> amount;              // ввод текущих данных
    total += amount;            // обработка текущих данных
    count++; }                  // изменение текущих данных (не забывайте!)
```

Было бы неплохо иметь возможность писать приложения, где сегмент кода (например, обработка транзакции) применяется к каждому элементу исходных данных нужное число раз, а размер набора данных для каждого выполнения программы разный. Заранее задавать размер набора данных, как в листинге 4.10, в таком случае не следует. Программа должна определять, когда обработан последний элемент набора данных. Один из способов решения этой задачи состоит в том, чтобы попросить пользователя указать, сколько элементов нужно обработать, и включить данное значение в условие цикла, однако пользователь не всегда доступен. Что если данные поступают по коммуникационному каналу с удаленного компьютера? Тогда первый элемент набора данных часто является счетчиком числа последующих элементов. Но и размер набора данных не всегда заранее известен или может быть очень большим. Одно дело считать пять элементов, другое — сотни и тысячи.

Наиболее распространенный подход к итеративной обработке — поочередный ввод данных, когда они доступны, и запрос у пользователя следующего элемента для обработки (либо анализ конца входного файла или опрос канала связи).

Это можно сделать двумя способами. Один состоит в том, чтобы использовать две отдельные переменные для хранения данных и ответа пользователя на вопрос о наличии других данных. После каждой транзакции пользователь отвечает на вопрос, есть ли еще данные для обработки. Второй способ — ввод специального значения (контрольного), указывающего приложению на завершение набора данных. Контрольное значение должно отличаться от возможных обычных данных. Для сумм транзакций это может быть отрицательное число или 0. Когда данные поступают по коммуникационной линии, таким значением является последнее значение в наборе данных.

Листинг 4.11 показывает реализацию цикла с контрольным значением 0 или отрицательным числом.

Листинг 4.11. Реализация цикла `while` с отрицательным или нулевым контрольным значением

```
#include <iostream>
using namespace std;

int main ()
{
  double total, amount; int count;
  total = 0.0; count = 0;                          // разная инициализация
  amount = 1.0;                                    // искусственный прием: почему 1, а не 10?
  while (amount > 0)                               // вычисление текущих данных
    {    cout << "Введите количество (для завершения - 0 или отрицательное число): ";
         cin >> amount;                            // ввод текущих данных
         total += amount;                          // обработка текущих данных
         count++;  }
  cout << "\nСумма по " << count << " транзакциям равна "<<total << endl;
  return 0;
}
```

В листинге 4.10 цикл продолжается до тех пор, пока `count` не превысит значения 5. Перед первым проходом цикла `count` равно 1, перед вторым — 2, перед пятым — 5, а после пятого, когда `count <= 5` принимает значение `false`, оно равно 6. Для программы из листинга 4.11 это не подходит. Здесь нужно, чтобы в конце выполнения значение `count` отражало число обработанных элементов. Вот почему `count` в листинге 4.11 инициализируется значением 0, а не 1.

Программист, использующий C++ (а на самом деле — любой программист), должен продумать этот вопрос при написании циклов. Корректны ли начальные значения? Обеспечивают ли они завершение цикла? Вот почему в данном примере число элементов так мало: это позволяет легко отследить итерации цикла. Вывод программы из листинга 4.11 показан на рис. 4.18.

```
Введите количество (для завершения - 0 или отрицательное число): 22
Введите количество (для завершения - 0 или отрицательное число): 33
Введите количество (для завершения - 0 или отрицательное число): 44
Введите количество (для завершения - 0 или отрицательное число): -1

Сумма по 4 транзакциям равна 98
```

Рис. 4.18. *Вывод программы из листинга 4.11 с отрицательным контрольным значением*

Но ведь число транзакций здесь некорректно! Перед тем как пользователь ввел –1.0, значение `count` было равно 3, и это правильно. После ввода –1.0 оно стало равно 4, что уже неправильно. Более того, это отрицательное число также добавляется к `total` и общая сумма тоже некорректна.

Такие технические проблемы можно решить несколькими способами, например, инициализировать счетчик значением –1 или уменьшать его после цикла. То же самое можно сделать с суммой `total`. Пусть код после цикла выглядит так:

```
count-; total -= amount;                          // коррекция после цикла
cout << "\nСумма по " << count << " транзакциям равна "
     <<total << endl;
```

Не очень элегантное решение, но оно работает. Другой вариант заключается в добавлении в середину цикла условного оператора и изменении значений total и count, когда значение amount отличается от контрольного:

```
while (amount > 0)                                    // вычисление текущих данных
  { cout << " Введите количество (отрицательное число или 0 - для завершения): ";
    cin >> amount;                                    // введите текущие данные
    if (amount > 0)                                   // проверка конца данных
       {   total += amount;                           // обработка текущих данных
           count++; } }
```

Как можно видеть, эти исправления устраняют проблемы за счет увеличения сложности кода. Не очень элегантно и часто (но не всегда) свидетельствует о проявлении концептуальной проблемы. И в самом деле, в решении, представленном в листинге 4.11, есть еще одна проблема, не техническая, а концептуальная. Проблема первого прохода цикла. Когда выделяется память для переменных C++, она содержит случайные значения (не совсем, правда, но об этом позднее). В некоторых случаях при выполнении программы такое значение может быть равно 0. Это означает, что программа будет завершаться без обработки данных. Чтобы предотвратить такой эффект, можно инициализировать amount некоторым значением — просто для предотвращения несвоевременного завершения цикла.

С точки зрения программной инженерии это некорректно. Здесь используется 1.0, что не имеет семантического смысла в контексте приложения. Если бы значение было равно 2.0, то результат был бы тем же. Данное значение не доносит до сопровождающего приложение программиста никакой мысли разработчика, а следовательно, только усложняет дело. Программист потратит какое-то время, пытаясь уяснить, что означает 1.0, прежде чем поймет, что ничего. Хорошо, если это займет немного времени. Все равно, такого рода огрехи только напрасно увеличивают сложность приложения.

Еще одна проблема с данным циклом в том, что контрольное значение не интерпретируется корректно. Когда пользователь вводит отрицательное число (или отрицательное значение поступает по коммуникационной линии), оно сначала добавляется к total и только после этого используется для завершения цикла.

Хорошее решение проблемы состоит в изменении структуры тела цикла. В листинге 4.11 тело цикла сначала считывает amount (это может быть контрольное значение) и обрабатывает его. Здесь предполагается, что предварительно нужно обработать введенное ранее (на предыдущей итерации) значение amount, и только в конце цикла считывается amount для следующей итерации. Структура цикла может выглядеть так:

```
while (amount > 0)                                    // оценка текущих данных
  { total += amount;                                  // обработка текущих данных
    count++;
    cout << "Введите количество (для завершения - 0 или отрицательное число): ";
    cin >> amount; }                                  // изменение текущих данных
```

А как насчет первой итерации цикла? Значение 1.0, использовавшееся в листинге 4.11, здесь не подходит. Следует ли инициализировать переменную значением 0? Если 0 добавиться к total, вреда не будет. Это возможно, но цикл завершится при первой проверке — условие amount > 0 даст false. Кроме того, такое решение не будет работать, если нужно вывести значение amount, которое вводилось или обрабатывалось каким-то другим нетривиальным образом.

Хорошим решением здесь станет так называемый метод *предварительного чтения*. Первое значение считывается перед циклом, обрабатывается в начале цикла, затем в конце цикла считывается новое значение, которое обрабатывается в начале цикла на следующей итерации. Данное решение показано в листинге 4.12, а результаты представлены на рис. 4.19.

Листинг 4.12. Реализация цикла `while` с предварительным чтением

```
#include <iostream>
using namespace std;

int main ()
{
  double total, amount; int count;
  total = 0.0; count = 0;                                    // разная инициализация
  cout << "Введите количество (для завершения - 0 или отрицательное число): ";
  cin >> amount;                                             // первый ввод текущих данных
  while (amount > 0)                                         // вычисление текущих данных
  { total += amount;                                         // обработка текущих данных
    count++;
    cout << "Введите количество (для завершения - 0 или отрицательное число): ";
    cin >> amount; }                                         // ввод текущих данных
  cout << "\nСумма по " << count << " транзакциям равна "<<total << endl;
  return 0;
}
```

```
Введите количество (для завершения - 0 или отрицательное число): 22
Введите количество (для завершения - 0 или отрицательное число): 33
Введите количество (для завершения - 0 или отрицательное число): 44
Введите количество (для завершения - 0 или отрицательное число): -1

Сумма по 3 транзакциям равна 99
```

Рис. 4.19. *Вывод программы из листинга 4.12 с предварительным чтением*

Все проблемы (и необходимость их исправления) исчезли. Переменные `count` и `total` инициализируются своими начальными значениями (0). Переменная `amount` не инициализируется — ее начальное значение (каким бы оно ни было) заменяется при операции ввода. После цикла нет никакой постобработки, которая настраивала бы некорректно измененные в цикле значения.

Недостаток такого решения в том, что операторы ввода присутствуют здесь в двух экземплярах. В реальной ситуации это не проблема, поскольку, если ввод и проверка исходных данных требуют нескольких операторов, их можно инкапсулировать в функцию. Данную функцию все равно потребуется вызвать дважды, но с этим можно смириться. Ничто не совершенно.

Прежде чем переходить к циклам `do-while`, следует проиллюстрировать некоторые моменты проектирования циклов `while`. Возьмем, например, потоковую обработку символов. Для простоты будем просто отображать символы на экране (эхо), подсчитывать их количество и число пробелов (если они есть). Обработка должна продолжаться, пока пользователь не нажмет клавишу `Enter` (символ `'\n'`). Будем использовать структуры цикла с предварительным чтением. Первый символ вводится перед циклом `while`, а в начале цикла на экране отображается введенный ранее символ. В условии цикла проверяется, является ли следующий символ символом новой строки. Если *нет* (условие цикла равно `true`), то обработка продолжается. В начале цикла символ выводится на экран, подсчитывается, а в конце цикла считывается следующий символ. Если это символ новой строки, то условие цикла даст `false` и цикл завершится.

Для ввода символа применяется функция `get()` из библиотеки `iostream`. Она передается как сообщение объекту `cin`. О синтаксисе сообщений рассказывалось в главе 2, и теперь полезно им воспользоваться. Фактически нужно знать лишь, что `cin.get()` возвращает следующий символ из буфера ввода.

Листинг 4.13. Цикл `while` с предварительным чтением для получения символов

```
#include <iostream>
using namespace std;

int main ()
{
  char ch; int count = 0, spaces = 0;          // инициализация счетчиков
  cout << "\nНаберите предложение и нажмите Enter\n";
  ch = cin.get();                              // предварительное чтение для цикла
  while (ch != '\n')                           // нет точки с запятой после условия
  { cout << ch;                                // обработка данных: эхо, проверка, подсчет
    if (ch == ' ')
      spaces++;
    count++;
    ch = cin.get(); }                          // смена текущих данных
  cout << "\nОбщее число символов " << count << endl;
  cout << "Число пробелов равно " << spaces << endl;
  return 0;
}
```

```
Наберите предложение и нажмите Enter
Это проверка
Это проверка
Общее число символов 15
Число пробелов равно 3
```

Рис. 4.20. *Результаты выполнения программы из листинга 4.13 (обработка вводимых символов)*

В листинге 4.13 показано решение данной проблемы, а на рис. 4.20 продемонстрированы результаты теста.

Здесь мы снова сталкиваемся с разницей между внешним видом и сутью вещей. Программа показывает, что первый введенный пользователем символ отображается перед вводом второго символа, а второй — перед вводом третьего и т. д. Но если выполнить программу, окажется, что символы не появятся на экране, пока не будет нажата клавиша Enter, после чего они отобразятся все сразу. Причина в том, что при вызове `cin.get()` происходит ввод не с клавиатуры, а из внутреннего буфера в память компьютера. Когда пользователь нажимает клавиши на клавиатуре, данные поступают в буфер и становятся доступными программе только при нажатии Enter или при заполнении буфера. Применение буферов может повысить производительность программы при частом обмене небольшими порциями данных с файлом. При буферизации медленные операции ввода-вывода с внешним файлом выполняются сразу для большого количества данных. (Сама операция занимает практически одно и то же время, независимо от объема данных.) Кроме того, обмен с буфером при вводе данных осуществляется намного быстрее (впрочем, для данной программы это не важно). Так что не удивляйтесь, если не увидите вывода при наборе данных.

Обратите внимание, что условие цикла `while (ch != '\n')` проверяется сразу за оператором `ch = cin.get()`. В первый раз это будет оператор перед циклом, а далее — операторы в конце цикла. Такая структура кода наводит на мысль об использовании известного стиля C++: комбинировании присваивания и проверки условия, что показано в листинге 4.14.

Это очень популярный метод в C++. В программе из листинга 4.12 нельзя было бы использовать оператор ввода (`cin >> amount;`), поскольку он не возвращает введенного пользователем значения. Он возвращает значение, но это значение объекта `cin`, а не символ. Оператор ввода из листинга 4.13 (`ch = cin.get();`) возвращает значение символа и может использоваться в программе, показанной в листинге 4.14.

Листинг 4.14. Цикл `while` с присваиванием в условии цикла

```
#include <iostream>
using namespace std;

int main ()
{
  char ch; int count = 0, spaces = 0;          // инициализация счетчиков
  cout << "\nНаберите предложение и нажмите Enter\n";
  while ((ch = cin.get()) != '\n')             // изменение текущих данных
  { cout << ch;                                // обработка данных: эхо, проверка, подсчет
    if (ch == ' ') spaces++;                   // ОК для одной строки
    count++; }
  cout << "\nОбщее число символов " << count << endl;
  cout << "Число пробелов равно " << spaces << endl;
  return 0;
}
```

Обратите внимание на круглые скобки, в которые заключен оператор ввода в листинге 4.14. Если их опустить, это не будет синтаксической ошибкой, но изменится смысл кода:

```
cout << "\nНаберите предложение и нажмите Enter\n";
while (ch = cin.get() != '\n')          // оператор ввода не заключен в скобки
{ cout << ch;
  if (ch == ' ')
    spaces++;
  count++; }
```

Важен приоритет операций. Операция присваивания имеет более низкий приоритет, чем сравнение на неравенство. Это означает, что компилятор будет видеть следующее:

```
while (ch = (cin.get() != '\n')      // совсем другая история
```

Здесь вводится символ, а затем он сравнивается с символом новой строки. Для всех символов (кроме последнего во вводимой строке) результатом будет `true` (они не являются символами новой строки), и переменная `ch` получает значение 1 (это непечатаемый код). Символы отображаются некорректно, а число пробелов в строке будет равно нулю.

Не следует жаловаться на неверный порядок операций в C++. Нужно знать этот порядок, и проблем не будет. В случае сомнения (и даже, если не сомневаетесь) используйте скобки.

Итерации в цикле do-while

Цикл `do-while` очень напоминает цикл `while`. Часто они взаимозаменяемы. Основная разница в том, что в цикле `do-while` условие проверяется в конце цикла, после каждой итерации, а в цикле `while` оно проверяется в начале цикла, перед итерацией.

Как и цикл `while`, цикл `do-while` управляет повторяющимся выполнением тела цикла, состоящего из одного оператора (или блока операторов в фигурных скобках). Цикл `do-while` имеет общую структуру:

```
предыдущий_оператор;
do
  оператор;                  // или { операторы }
while (выражение);
следующий_оператор;
```

После выполнения предыдущего_оператора обрабатывается тело цикла после ключевого слова do. Затем вычисляется выражение цикла. Если оно дает true, то тело цикла выполняется снова. Если при вычислении выражения получается false, то итерации прерываются и выполняется следующий_оператор.

Такая структура обеспечивает как минимум однократное выполнение тела цикла. Чтобы завершить цикл и предотвратить бесконечные итерации, в теле цикла должно изменяться выражение цикла.

Во избежание путаницы программисты часто заключают тело цикла в фигурные скобки, даже если оно содержит лишь один оператор:

```
do
  { оператор; }
while (выражение);
```

Чтобы избежать синтаксической ошибки, в цикле do-while после выражения цикла нужно поставить точку с запятой (в отличие от цикла while, где она как раз не нужна). Следует понимать, что размещение точки с запятой после выражения цикла в цикле while не смущает компилятор и не приводит к синтаксической ошибке, но код становится некорректным (семантическая ошибка). Чтобы указать сопровождающему приложение программисту на особый смысл ключевого слова while и на необходимость использования точки с запятой в конце строки, некоторые разработчики помещают закрывающую фигурную скобку на ту же строку, где находится ключевое слово while.

```
do
  { оператор;
} while (выражение);          // фигурная скобка предупреждает о присутствии
                              // точки с запятой
```

Построение тела цикла здесь аналогично циклу while:

```
инициализация_текущих_данных;
do { изменение_текущих_данных;
     обработка_текущих_данных;
     } while (вычисление_текущих_данных);
```

Для примера обработки транзакций и вычисления суммы компоненты такой структуры должны выполнять следующие операции:

```
инициализация_текущих_данных:     установка total и count в 0
изменение_текущих_данных:         ввод нового значения amount
обработка_текущих_данных:         если положительное, увеличить total и count
вычисление_текущих_данных:        сравнить amount с контрольным значением
```

Данная версия программы показана в листинге 4.15. Ее вывод будет соответствовать рис. 4.19.

Листинг 4.15. Реализация цикла do-while без предварительного чтения

```
#include <iostream>
using namespace std;

int main ()
{
  double total, amount; int count;
  total = 0.0; count = 0;                                  // инициализация текущих данных
do {
  cout << "Введите количество (для завершения - 0 или отрицательное число): ";
  cin >> amount;                                           // первый ввод текущих данных
```

```
    if (amount > 0)                                    // проверка на конец данных
    { total += amount;                                 // обработка текущих данных
      count++;
    } while (amount > 0)                               // вычисление текущих данных
    cout << "\nСумма по " << count << " транзакциям равна "<<total << endl;
    return 0;
}
```

Как можно видеть, с одной стороны цикл do-while упрощает инициализацию и устраняет необходимость предварительного чтения (сравните с листингом 4.12), с другой — нужны дополнительные операции в середине цикла, чтобы избежать ошибочной обработки контрольного значения как обычных вводимых данных.

Подсчет содержащихся во вводимой строке символов пробела можно делать в цикле do-while, как показано в листинге 4.16. Использование цикла do-while позволяет обойтись без предварительного чтения. Аналогично предыдущему примеру из листинга 4.15, эта структура требует проверки в теле цикла ввода контрольного значения. Следовательно, вычисление текущих данных выполняется дважды — в теле цикла и в логическом условии цикла.

Листинг 4.16. Цикл do-while для ввода символов

```
#include <iostream>
using namespace std;

int main ()
{
   char ch; int count = 0, spaces = 0;                 // инициализация данных
   cout << "\nНаберите предложение и нажмите Enter\n";
do {
    ch = cin.get();                                    // изменение текущих данных
    if (ch != '\n')                                    // проверка текущих данных
     { cout << ch;
       if (ch == ' ') spaces++;                        // обработка текущих данных
       count++; }
     } while (ch != '\n');                             // вычисление текущих данных
   cout << "\nОбщее число символов " << count << endl;
   cout << "Число пробелов равно " << spaces << endl;
   return 0;
}
```

Здесь значение символа ch устанавливается в присваивании и проверяется в операторах условия. Это открывает возможность комбинирования присваивания и проверки в одном операторе, как в листинге 4.17.

Листинг 4.17. Цикл do-while с присваиванием в операторе условия

```
#include <iostream>
using namespace std;

int main ()
{
   char ch; int count = 0, spaces = 0;                 // инициализация данных
   cout << "\nНаберите предложение и нажмите Enter\n";
```

```
    do {
      if ((ch = cin.get()) != '\n')                    // изменение текущих данных
      { cout << ch;
        if (ch == ' ') spaces++;                       // обработка текущих данных
        count++; }
      } while (ch != '\n');                            // вычисление текущих данных
    cout << "\nОбщее число символов " << count << endl;
    cout << "Число пробелов равно " << spaces << endl;
    return 0;
}
```

Данная оптимизация также не влияет на производительность программы или ее корректность, но программный код становится более лаконичным и элегантным.

Итерации с циклом for

Цикл for подходит в тех случаях, когда число итераций известно заранее, до начала цикла. Это не очень важный фактор. Визуально данная форма итерации представляется как совмещение трех наиболее значимых элементов архитектуры цикла: инициализация текущего значения перед первой итерацией, вычисление текущего значения перед началом следующей итерации и изменение текущего значения после итерации (перед началом следующей). В других циклах эти элементы распределены по разным местам цикла.

Цикл for имеет следующую стандартную форму, где в скобках комбинируются три выражения, управляющие выполнением тела цикла (инициализация текущих данных, их вычисление и модификация). Эти выражения разделяются двоеточиями (что еще раз убеждает — изучение C++ никогда не превратится в скучное занятия). Вот почему последнее выражение не имеет точки с запятой перед закрывающей скобкой.

```
предыдущий_оператор;
for (начальное_выражение; продолж_выражение; инкр_выраж)
  оператор;        // можно использовать составной оператор в фигурных скобках
следующий_оператор;
```

Начальное_выражение вычисляется только один раз, перед первой итерацией. Здесь удобно инициализировать значения для работы цикла: индекс, счетчик элементов, сумму и пр.

Инкр_выраж вычисляется в конце следующей итерации, непосредственно после выполнения тела цикла. Это общепринятое место изменения текущих данных, приращения индексов, счетчиков, меток и т. д.

Продолж_выражение вычисляется перед первой итерацией и перед каждой последующей итерацией. Данное выражение позволяет оценить необходимость следующей итерации цикла. Если при его вычислении получается true, то выполняется оператор цикла, а для определения необходимости следующей итерации снова вычисляются инкр_выраж и продолж_выражение. Если же в результате вычисления продлож_выражения будет получено false, то выполнение оператора цикла будет завершено.

Нужно понимать, что цикл for эквивалентен следующему циклу while:

```
предыдущий_оператор;
начальное_выражение;
while (продолж_выражение)
  { оператор;                  // или последовательность операторов
    инкр_выраж; }
следующий_оператор;
```

В листинге 4.18 показан пример обработки транзакций с использованием цикла for. Инициализация включает в себя установку значения count в 0, а проверка на продолжение предусматривает проверку контрольного значения (вот почему предварительное чтение все же необходимо). Инкремент — это увеличение переменной count.

Листинг 4.18. Реализация обработки транзакций с использованием цикла for

```
#include <iostream>
using namespace std;

int main ()
{
  double total, amount; int count;
  total = 0.0;                                                    // разная инициализация
  cout << "Введите количество (для завершения - 0 или отрицательное число): ";
  cin >> amount;                                                  // ввод текущих данных
  for (count=0; amount>0; count++)                                // три выражения
  { total += amount;                                              // обработка текущих данных
    cout << "Введите количество (для завершения - 0 или отрицательное число): ";
    cin >> amount; }                                              // изменение текущих данных
  cout << "\nСумма по " << count << " транзакциям равна " <<total << endl;
  return 0;
}
```

Каждая из трех составляющих в цикле for представляет собой выражение. Это глубокомысленное наблюдение означает, что в каждом таком выражении может использоваться последовательность разделенных запятыми выражений. Не забывайте, что запятая — полноправная операция C++. Ее операнды (выражения) вычисляются слева направо, и возвращается самое правое значение. В цикле for возвращаемые значения не важны — они отбрасываются. Единственное исключение — продолж_выражение, которое определяет, нужна ли следующая итерация. Это означает, что начальное_выражение можно расширить, как показано в листинге 4.19.

Листинг 4.19. Цикл for **с операцией "запятая" в начальных выражениях**

```
#include <iostream> using namespace std;

int main ()
{
  double total, amount; int count;                                // нет инициализации
  cout << "Введите количество (для завершения - 0 или отрицательное число): ";
  cin >> amount;                                                  // ввод текущих данных
  for (total = 0.0; count=0; amount>0; count++)
    { total += amount;
      cout << "Введите количество (для завершения - 0 или отрицательное число): ";
      cin >> amount; }
  cout << "\nСумма по " << count << " транзакциям равна " <<total << endl;
  return 0;
}
```

Еще более интересна программа с вводом и обработкой символов, показанная в листинге 4.20. Здесь в начальном_выражении используются выражения, разделенные запятыми, а присваивание используется как часть сравнения в продолж_выражении (сравните эту версию с листингами 4.13 и 4.16).

Листинг 4.20. Цикл `for` с присваиванием в продолж_выражении

```
#include <iostream>
using namespace std;

int main ()
{
  char ch; int count, spaces;                    // нет инициализации
  cout << "\nНаберите предложение и нажмите Enter\n";
  for (count=0, spaces=0; (ch=cin.get())!='\n'; count++)
   { cout <<ch;                                  // обработка следующего вводимого символа
     if (ch == ' ') spaces++; }
  cout << "\nОбщее число символов " << count << endl;
  cout << "Число пробелов равно " << spaces << endl;
  return 0;
}
```

Еще один интересный момент — возможность определения переменных в начальном_выражении цикла `for`. Данная программа складывает квадраты первых натуральных чисел. Цикл `for` инициализирует переменную `n` значением 1, проверяет, достигает ли она предела `num` и увеличивает `n` после каждой итерации. Так как переменная `n` используется в цикле только один раз, нет никакой необходимости задавать для нее более широкую область действия. Вот почему она определяется в операторе `for`, а не в функции `main()`. Это популярный принцип C++. Результат тестового выполнения программы показан на рис. 4.21.

```
Введите, сколько нужно сложить квадратов: 4
Сумма квадратов равна 30
```

Рис. 4.21. *Результат выполнения программы из листинга 4.21 (сложение натуральных чисел)*

Листинг 4.21. Вычисление суммы квадратов с помощью цикла `for`

```
#include <iostream>
using namespace std;

int main ()
{
  int sum=0, num;
  cout "\nВведите, сколько нужно сложить квадратов: ";
  cin >> num;
  for (int n = 1; n <= num; n++)
    { sum += n * n; }
  cout << "Сумма квадратов равна " << sum << endl;
  return 0;
}
```

Как уже было показано в листингах 4.19 и 4.20, C++ позволяет программисту инициализировать несколько переменных в начальном_выражении цикла `for`. Кроме того, C++ дает возможность определять в начальном выражении несколько переменных, если все они одного типа. В листинге 4.22 переменные `n` и `sum` определяются в цикле. К тому же, переменная `sum` обновляется в продолж_выражении — в качестве инструмента используется операция-запятая. Результат выполнения данной версии программы будет таким же, как для рис. 4.21. Как видно, тело цикла вырождается в пустой оператор.

Листинг 4.22. Цикл `for`, генерирующий пустой оператор

```
#include <iostream>
using namespace std;

int main ()
{
  int num;
  cout >> "\nВведите, сколько нужно сложить квадратов: ";
  cin >> num;
  for (int sum = 0, n = 1; n <= num; sum+=n*n; n++);          // !!
  cout << "Сумма квадратов равна " << sum << endl;
  return 0;
}
```

Многим программистам не нравится использовать точку с запятой в конце оператора цикла. Это не вполне обычное место для нее, что может запутать программиста, занимающегося сопровождением. Чтобы донести знания разработчика до такого программиста, они помещают точку с запятой на отдельную строку, например:

```
for (int sum = 0, n = 1; n <= num; sum+=n*n, n++)
;                                                          // !!
```

Другие программисты вовсе избегают пустых операторов, поскольку они вносят путаницу, а применяют вместо них структуру, аналогичную показанной в листинге 4.21, где в теле цикла содержится хотя бы один оператор. Самая большая проблема программы из листинга 4.21 — ее переносимость. Переменная `sum` определяется в цикле, но используется после его завершения. "Оригинал" C++ допускает это, но новый стандарт C++ интерпретирует такие действия, как синтаксическую ошибку: в цикле можно определять только переменные, используемые в самом цикле, но не вне его. Большинство компиляторов такой код компилируют, но применять данный метод не следует: обычно не стоит увлекаться оптимизацией циклов `for`.

Операторы перехода в C++

Условные операторы и операторы цикла — неотъемлемый инструмент программирования. Без них нельзя написать даже простейшую программу. Они — важнейшая необходимость. Другие операторы управления полезны, но не столь необходимы. Они представляют синтаксические дополнения, делающие программу толково написанной и более эстетически привлекательной.

Такими операторами C++ являются различные виды переходов. Разработчики программ любят переходы, поскольку они позволяют им результативно и эффективно передавать управление в любое место исходного кода программы. Между тем, программу с переходами труднее анализировать, чем программу без оных. Чтобы понять результат выполнения программы с последовательными операторами, сопровождающему ее программисту нужно разобраться только в тех операторах, которые непосредственно предшествуют анализируемому. Когда управление передается на этот оператор из других мест программы, на работу оператора влияют все такие участки кода. Анализ программы усложняется. Вот почему у переходов плохая репутация.

В C++ делается попытка предложить компромиссный вариант. Он допускает переходы, чтобы программисты могли писать более мощный и лаконично составленный исходный код. Но переходы здесь ограничены, чтобы при сопровождении программы не пришлось решать слишком сложные задачи.

Оператор break

Этот оператор используется для немедленного выхода из цикла. После выполнения данного оператора управление передается на оператор, следующий за циклом. Выход из цикла в середине его выполнения — шаг, конечно, решительный, но он не слишком осложняет управляющую структуру. Оператор break нельзя использовать для выхода из ветви оператора if (в противном случае управляющая структуру стала бы чрезмерно запутанной). Позднее мы рассмотрим применение оператора break внутри операторов switch.

Нет никакого смысла выполнять оператор break безусловно. Это означает, что цикл не работает вовсе. Обычно break выполняется в операторе условия. В логическом выражении этого условия задается условие прекращения цикла. Часто такой метод позволяет упростить условие цикла. Например, можно просто построить "бесконечный" цикл с выходом по break.

Рассмотрим листинг 4.12 с циклом, обрабатывающим вводимые данные до получения контрольного значения. В условии цикла используется значение переменной amount. Следовательно, данная переменная должна инициализироваться перед циклом, а это задача предварительного чтения.

```
cout << "Введите количество (для завершения - 0 или отрицательное число): ";
cin >> amount;                                          // ввод текущих данных
while (amount > 0)                                      // оценка текущих данных
{ total += amount;                                      // обработка текущих данных
  count++;
  cout << "Введите количество (для завершения - 0 или отрицательное число): ";
  cin >> amount;}                                       // изменение текущих данных
```

Использование оператора break позволяет заменить условие цикла на нечто неизменяемое, например на while (1 == 1). Так как это условие всегда равно true, при записи данного выражения гораздо меньше шансов сделать ошибку. Поскольку в условии не нужно применять значение amount, нет нужды в предварительном чтении. Значение amount можно получать в начале цикла, а не в конце. Проблема этой структуры цикла — прекращение цикла при получении контрольного значения. Решение дает оператор break. Итерации нужно продолжать, пока amount > 0. Следовательно, условием прекращения цикла является отрицание этого, т. е. amount <= 0.0. Когда это (отрицательное) условие содержит true, выполняется оператор break и управление передается из цикла на следующий оператор:

```
while (1 == 1)                                          // бесконечный цикл
{ cout << "Введите количество (для завершения - 0 или отрицательное число): ";
  cin >> amount;                                        // ввод текущих данных
  if (amount <= 0.0) break;                             // явный выход
  total += amount;                                      // обработка текущих данных
  count++; }
```

Можно возразить, что перенос проверки amount из выражения цикла на оператор break не особенно влияет на сложность программы. Но зато позволяет устранить предварительное чтение.

В версии данного алгоритма с циклом do-while (см. листинг 4.15) не используется предварительное чтение, но дважды проверяется, содержит ли следующее вводимое число допустимое значение (в середине и в конце цикла):

```
do {
  cout << "Введите количество (для завершения - 0 или отрицательное число): ";
  cin >> amount;                                        // первый ввод текущих данных
  if (amount > 0)                                       // проверка на конец данных
  {  total += amount;                                   // обработка текущих данных
```

```
    count++;
  } while (amount > 0);                    // вычисление текущих данных
```

Применение оператора `break` позволило заменить условие цикла на тривиальное условие, которое всегда равно `true` (типа `1 == 1` или даже просто `1`). Для завершения цикла при появлении контрольного значения условие `amount > 0` нужно отрицать, аналогично предыдущему примеру. Когда `amount <= 0`, оператор `break` передает управление на следующий за циклом оператор. Структура цикла в чем-то упрощается — нет необходимости в локальном составном операторе с его фигурными скобками. Обратите внимание, что `while (1)` — вполне законная альтернатива `while (1 == 1)`, так как в C++ ненулевое значение всегда равно `true`:

```
do {
  cout << "Введите количество (для завершения - 0 или отрицательное число): ";
  cin >> amount;                  // первый ввод текущих данных
  if (amount <= 0) break;         // проверка текущих данных
  total += amount;                // обработка текущих данных
  count++;                        // не нужен составной оператор
} while (1)                       // не требуется вычисление здесь текущих данных
```

Вот еще один пример использования оператора `break` в проверяемом диапазоне, часто применяемого для контроля допустимости ввода. Предположим, что пользователю нужно ввести ответ в диапазоне от 1 до 5. Если пользователь делает ошибку, ввод должен повторяться, пока не будет введено допустимое значение. В листинге 4.23 используется цикл `do-while`, так как тело цикла должно выполняться хотя бы один раз. Переменная `error_flag` устанавливается в 1, если вводимое значение недопустимо, и в 0, если оно попадает в диапазон.

Листинг 4.23. Использование цикла `do-while` для проверки допустимости ввода

```
#include <iostream>
using namespace std;
const int N = 5;

int main ()
{
  int num, error_flag;
  do {
    cout << "Введите число между 1 и " << N << ": ";
    cin >> num;
    if   (num < 1 || num > N)
      {  cout << "Ввод некорректен, повторите\n";
         error_flag = 1; }
    else
        error_flag = 0;
     }  while (error_flag == 1);
  cout << "Вы ввели " << num << endl;
  return 0;
}
```

Это популярный метод для коммуникаций между разными частями программы. В одной части программы (условии цикла) нужно знать, что произошло в другой ее части (теле цикла). Чтобы такое было возможным, в одной части программы проверяется переменная, устанавливаемая в другой ее части.

Некоторые программисты предпочитают не заводить большого количества флагов или других управляющих переменных, так как единственное их назначение — передавать информацию из одной части программы в другую. Это увеличивает связность программного кода и его сложность. Другим способом реализации данного алгоритма является повторение теста в условии цикла (вместо использования флага ошибки):

```
do {
  cout << "Введите число между 1 и " << N <<": ";
  cin >> num;
  if (num < 1 || num > N)
    cout << "Ввод некорректен, повторите\n";
  } while (num < 1 || num > N);
```

Это более разумное решение. Еще один подход состоит в применении бесконечного цикла и прерывании его выполнения, когда значение допустимое. Следовательно, цикл продолжается с запросом данных, когда n < 1 или num > N, и завершается, когда это условие принимает значение false.

```
do {
  cout << "Введите число между 1 и " << N <<": ";
  cin >> num;
  if (!(num < 1 || num > N)
    cout << "Ввод некорректен, повторите\n";
  } while (true);
```

Отметим, что это третья форма "бесконечного" цикла, где литеральное значение (true) используется в качестве условия его продолжения.

Многие программисты предпочитают явное отрицание условия составного оператора. Для этого можно заменить каждую операцию && на операцию ||, а каждую операцию || на && и применять отрицание отдельных условий. Рассмотрим, например, выражение a1 && (~a2) || a3, где a1, a2 и a3 — булевы выражения. Его отрицанием будет (~a1) || a2 && (~a3). В нашем случае отрицание для прерывания цикла выглядит так:

```
do {
  cout << "Введите число между 1 и " << N <<": ";
  cin >> num;
  if (num >=1 && num <= N) break;                    // просто и красиво
    cout << "Ввод некорректен, повторите\n";
  } while (true);
```

Некоторые программисты ощущают дискомфорт при отрицании составных условий, но это очень полезный навык, и в нем надо по возможности хорошо попрактиковаться.

Оператор break — один из достаточно простых и эффективных переходов в C++. Другие переходы либо более опасны, либо не так эффективны.

Оператор continue

Оператор continue — еще более простая модификация оператора break. Как и break, он используется в конструкциях циклов и пропускает остальную часть тела цикла, заключенную между оператором continue и его концом.

Оператор continue можно использовать в циклах while, do-while и for. В циклах while и do-while он переходит к концу или началу цикла для проверки условия цикла. В цикле for оператор continue не обходит выражение инкремента, а только пропускает остальную часть тела цикла.

Рассмотрим, например, листинг 4.11 (без предварительного чтения) и его модификацию, которая решает проблемы за счет обновления текущих данных, когда контрольное значение еще не введено.

```
while (amount > 0)                          // вычисление текущих данных
  { cout << "Введите количество (для завершения - 0 или отриц.число): ";
    cin >> amount;                          // изменение текущих данных
    if (amount > 0) continue                // проверка допустимости данных
       {   total += amount;                 // обработка текущих данных
           count++; } }
```

Вместо использования блока в операторе условия можно отрицать условие и применить оператор continue:

```
while (amount > 0)                          // вычисление текущих данных
  { cout << "Введите количество (для завершения - 0 или отриц.число): ";
    cin >> amount;                          // изменение текущих данных
    if (amount <= 0) continue               // проверка допустимости данных
    total += amount;                        // обработка текущих данных
    count++; }
```

Улучшение не очень значимое. Как уже отмечалось, оператор continue весьма примитивен. Его всегда можно заменить условным оператором. Нечасто приходится видеть оператор continue, значительно улучшающий исходный код. На самом деле такой пример встретился мне лишь однажды, и я даже подумал, что его стоит записать, но что-то отвлекло меня, и теперь я не могу привести этот пример.

Оператор goto

Оператор goto — король переходов. Именно неограниченное применение этих операторов вызывало столько дискуссий: шли споры вокруг того, вредны ли такие переходы и следует ли объявить их вне закона. Действительно, во многих современных языках программирования запрещены неограниченные переходы. И в C++ тоже.

C++ допускает переходы goto только в пределах одной функции. Это означает, что и оператор goto, и целевой оператор (куда выполняется переход) обязаны находиться в одной функции. Кроме того, не допускаются переходы через определения. Достаточно ограничительная мера для обычного перехода. Вот почему в C++ оператор goto менее вреден, чем в других языках. Вот как он здесь выглядит:

```
void foo
{ ...
  goto label1;          // нет двоеточия после имени метки
  int x;                // переход через определение: - синтаксическая ошибка
    ...
  label1: оператор;     // двоеточие после имени метки
    ...
  goto label1; }        // нет перехода через определение: OK
```

Метка — это идентификатор. Она помещается перед оператором, куда передается управление, и в конце оператора goto. Имена меткам придумывает программист. В отличие от идентификаторов переменных, функций и типов, метки определять не нужно. Они просто используются. Идентификаторы меток имеют свое собственное пространство имен. Это означает, что их имена не конфликтуют с другими идентификаторами: именами переменных, функций или типов.

За меткой, применяемой в качестве цели перехода, ставится двоеточие. В метке, которая указывается в операторе goto, двоеточие не требуется — оператор заканчивается точкой с запятой.

В листинге 4.24 приведен пример обработки транзакций, реализованный без циклов — только с операторами условия и переходами. Здорово, не правда ли?

Листинг 4.24. Обработка транзакций с помощью переходов `goto`

```
#include <iostream>
using namespace std;

int main ()
{
  double total=0.0, amount; int count=0;                    // инициализация
start:
  cout << "Введите количество (для завершения - 0 или отрицательное число): ";
  cin >> amount;                                            // ввод (изменение) текущих данных
  if (amount <= 0) goto finish;                             // вычисление текущих данных
  total += amount;                                          // обработка текущих данных
  count++;
  goto start;                                               // переход к началу цикла
finish:
  cout << "\nСумма по 5 транзакциям равна "<<total << endl;
  return 0;
}
```

Некоторые программисты, особенно разрабатывающие программы с помощью блок-схем, любят данный стиль программирования. В таком небольшом примере, вероятно, неважно, что именно использовать — переходы или циклы. Однако обычно полезно избегать применения операторов `goto`. Используйте их только в том случае, если это дает явные (и важные) преимущества.

Переходы return и exit

Оператор `return` представляет переход, который завершает выполнение функции. Если это функция `main()`, то программа завершается. Если это некая другая функция, вызываемая из `main()` прямо или косвенно, то данная вызываемая функция завершает работу и управление возвращается вызвавшей ее функции.

Когда функция возвращает тип, отличный от `void`, она обязана иметь оператор `return`. Если возвращаемый тип есть тип `void`, то оператор `return` не обязателен.

Оператор `return` может иметь или не иметь аргументы. Если функция определена как `void`, то в `return` не должно быть аргументов. C++ унаследовал из C две формы функции `main()`: одну с возвращаемым типом `int`, другую — с типом `void`. В новом стандарте C++ предпочтение отдается первой форме, однако можно встретить немало унаследованного кода C++, в котором функция `main()` не имеет возвращаемого типа, а следовательно, `return` в ней не используется. Если в функцию `void main()` включить необязательный оператор `return`, то он может выглядеть так:

```
void main(void)
{ . . .
  return; }.            // нет аргумента, нет круглых скобок
```

Когда оператор `return` используется в функции `void`, он не должен иметь ни аргумента, ни скобок: `return 0` будет ошибкой, `return()` тоже.

В предыдущих примерах использовалась функция `main()`, возвращающая целочисленное значение. Как и любая отличная от `void` функция, функция `main()` должна иметь оператор `return` и он должен возвращать целочисленное значение

(или значение, которое можно преобразовать в целое). Функция main() выглядит так:

```
int main(void)
{ . . .
  return 0; }          // аргумент обязателен, а круглые скобки - нет
```

Компиляторы C++ должны старательно воспринимать следующую форму функции main():

```
main(void)             // по умолчанию возвращается тип int
{ . . .
  return 0; }          // круглые скобки - необязательны
```

Как уже упоминалось, в C++ пропуск информации о возвращаемом типе не означает, что тип возвращаемого значения (void). Это означает тип int, что соответствует унаследованной из C философии, согласно которой лаконично составленная программа лучше, чем программа не столь эффектная. Следовательно, нужно помочь программисту писать эффектные программы (опуская возвращаемый тип), а для этого предусматриваются специальные средства языка.

Позднее эта философия была вытеснена мнением, что такие программы заставляют сопровождающего ПО программиста тратить больше времени и усилий на ее понимание, т. е. программа кажется ему более сложной. Программист, желающий писать эффектные программы, должен подумать в первую очередь о читабельности и понятности исходного кода, в частности для тех, кто не имеет достаточного опыта и уровня подготовки.

Оператор return делает возвращаемое значение доступным для вызывающей функции. Например, вызов функции cin.get(), использованной в предыдущих примерах, делает значение символа доступным для уже обсуждавшихся алгоритмов. Это означает, что в функции get() есть некий оператор вида return c;, где c — имя переменной (оно может быть и другим) типа char.

Когда функция main() возвращает значение, его воспринимает операционная система и оно позволяет судить о нормальном или ненормальном завершении программы. На многих платформах возвращаемое программой значение отбрасывается.

Это не означает, что можно опустить оператор return, если разрабатываемая функция (включая функцию main()) возвращает типизированное значение. Вы взяли на себя обязательство (отличный от void возвращаемый тип), и с этим придется жить. Если тип возвращаемого функцией значения не определен как void, то функция должна иметь оператор return, возвращающий значение (выражение) соответствующего типа. Скобки вокруг выражения в return не обязательны (но используются).

На количество операторов return никаких ограничений нет. Если return выполняется в середине функции, то остальная часть функции просто игнорируется.

Рассмотрим упрощенный пример калькулятора, который просит пользователя ввести два операнда и операцию, а затем выводит результат операции (листинг 4.25). Для демонстрации здесь используется функция main(), не возвращающая значения. Результат выполнения программы представлен на рис. 4.22.

```
Введите операнд, операцию и другой операнд: 22/0
Деление на ноль
```

Рис. 4.22. *Результат выполнения программы из листинга 4.25 (деление на ноль)*

Листинг 4.25. Упрощенный калькулятор

```
#include <iostream>
using namespace std;

void main(void)
{
  double op1, op2; char ch;
  cout << "Введите операнд, операцию и другой операнд: ";
  cin >> op1 >> ch >> op2;
  if (ch == '*')
    cout << "Результат равен " << op1 + op2 << endl;
  else
    if (ch == '*')
     cout << "Результат равен " << op1 * op2 << endl;
    else
     if (ch == '-')
        cout << "Результат равен " << op1 - op2 << endl;
     else
        if (ch == '/')
           if (op2 ! = 0.0)
              cout << "Результат равен " << op1 / op2 << endl;
           else
              cout << "Деление на ноль" << endl;
        else
           cout << "Недопустимая операция" << endl;
}
```

Эта примитивная программа выполняет только четыре арифметических операции и не запоминает результата. Однако для данного обсуждения ее функциональности вполне достаточно. Конечно, такая программа не выполняет всех необходимых действий, которые должна делать программа для проверки допустимости ввода. Сейчас данная проверка выходит за рамки нашего обсуждения.

Во-первых, стоит обсудить форматирование. Код программы представляет собой условный оператор с глубоким уровнем вложенности. Для каждого уровня вложенности задается отступ на пару пробелов вправо. Форматирование достаточно хорошо показывает, из чего состоит код (вложенные условные операторы), однако он не подчеркивает (для сопровождающего программиста) выполняемые действия.

В данной программе выбирается один из пяти альтернативных вариантов (сложение, умножение, вычитание, деление, недопустимая операция), но структура кода не передает намерений разработчика тому, кто сопровождает программу.

Ниже приведена другая версия, в которой условные операторы отражают логику обработки:

```
if (ch == '+')                              // первый случай
  cout << "Результат равен " << op1 + op2 << endl;
else if (ch == '*')                         // второй случай
  cout << "Результат равен " << op1 * op2 << endl;
else if (ch == '-')                         // третий случай
  cout << "Результат равен " << op1 - op2 << endl;
else if (ch == '/')                         // четвертый случай; более сложный
  { if (op2 != 0.0)
     cout << "Результат равен " << op1 / op2 << endl;
    else
     cout << "Деление на ноль" << endl;}
```

```
else
  cout << "Недопустимая операция" << endl;
```

Операторы return могут превратить отдельные ветви обработки в независимые условные операторы, следующие один за другим без ключевых слов else:

```
if (ch == '+')                                  // первый случай
  { cout << "Результат равен " << op1 + op2 << endl; return; }
if (ch == '*')                                  // второй случай
  { cout << "Результат равен " << op1 * op2 << endl; return; }
if (ch == '-')                                  // третий случай
  { cout << "Результат равен " << op1 - op2 << endl; return; }
if (ch == '/')                                 // четвертый случай: более сложный
  { if (op2 != 0.0)
     cout << "Результат равен " << op1 / op2 << endl;
    else
     cout << "Деление на ноль" << endl;
    return;}
cout << "Недопустимая операция" << endl;
```

Еще один популярный метод завершения — вызов функции exit(), входящей в стандартную библиотеку stdlib.h. Хотя функцию завершает только оператор return (если это функция main(), то он завершает и программу), вызов exit() позволяет завершить программу вне зависимости от того, какая функция сделала этот вызов. Функция exit() вызывается с одним целочисленным аргументом. Согласно популярному соглашению, 0 означает нормальное завершение, а 1 — ненормальное. С помощью этих значений программа передает ОС информацию о способе завершения.

Чтобы защитить программу, желающую таким способом общаться с ОС, от будущих изменений в соглашениях, в файле stdlib.h (или, согласно новому стандарту, cstdlib) определяются две символические литеральные константы: EXIT_SUCCESS и EXIT_FAILURE. Их рекомендуется использовать вместо кодов возврата 0 и 1. Вот пример применения этих констант в приведенной выше программе:

Листинг 4.26. Вызовы библиотечной функции exit()

```
#include <iostream>
#include <cstdlib>
using namespace std;

void main(void)
{
  double op1, op2; char ch;
  cout << "Введите операнд, операцию и другой операнд: ";
  cin >> op1 >> ch >> op2;
  if (ch == '+')                                    // первый случай
    cout << "Результат равен " << op1 + op2 << endl;
  else if (ch == '*')                               // второй случай
    cout << "Результат равен " << op1 * op2 << endl;
  else if (ch == '-')                               // третий случай
    cout << "Результат равен " << op1 - op2 << endl;
  else if (ch == '/')                               // четвертый случай: более сложный
    { if (op2 != 0.0)
        cout << "Результат равен " << op1 / op2 << endl;
      else
        cout << "Деление на ноль" << endl;}
```

```
  else                                                    // пятый случай: ошибка
    { cout << "Недопустимая операция" << endl;
      exit(EXIT_FAILURE); }                               // провал
  exit(EXIT_SUCCESS);                                     // все OK
}
```

На самом деле эти библиотечные константы имеют значения 0 и 1. Идея в том, что в один прекрасный день по каким-то причинам в операционной системе может быть введен другой набор значений, ожидаемых от программы C++. Тогда литералы 0 и 1 приведут к проблемам — ОС их не поймет. В то же время значения библиотечных констант EXIT_SUCCESS и EXIT_FAILURE легко исправить, и программа, использующая данные имена, снова будет общаться с операционной системой корректно. Вынужденная логика, но многие программисты применяют эти константы.

Оператор switch

Оператор switch — это инструмент для принятия в программе многозначных решений, позволяющих выполнять одну из многих ветвей. Альтернативные пути выбираются в зависимости от значения целочисленного выражения — выражения в круглых скобках за ключевым словом switch. Остальная часть оператора состоит из ветвей, заключаемых в фигурные скобки (открывающая и закрывающая фигурные скобки обязательны).

Каждая ветвь состоит из ключевого слова case, значения того же типа, что и выражение switch, двоеточия и набора из одного или более операторов, завершающихся точкой с запятой:

```
switch (выражение) {                                      // фигурные скобки обязательны
  case конст_выраж1: операторы;                           // первая ветвь
  case конст_выраж2: операторы;                           // другие ветви
  ...
  default: операторы;                                     // ветвь по умолчанию
}
```

Выражение в операторе switch может быть только типа char, short, int или long (целые типы). Типы с плавающей точкой (float, double или long double) не допускаются. Нельзя использовать и типы, определяемые программистом: структуры, массивы или классы.

Метки в case должны быть выражениями-константами, т. е. константами этапа компиляции, и того же типа, что и выражение switch.

Например, упрощенный калькулятор из листинга 4.25 можно реализовать с помощью оператора switch:

Листинг 4.27. Калькулятор с оператором switch (плохая программа)

```
#include <iostream>
#include <cstdlib>
using namespace std;

void main(void)
{
  double op1, op2; char ch;
  cout << "Введите операнд, операцию и другой операнд: ";
  cin >> op1 >> ch >> op2;
```

```
    switch(ch) {                                          // обязательные фигурные скобки
      case '+': cout << "Результат равен " << op1 + op2 << endl;
      case '*': cout << "Результат равен " << op1 * op2 << endl;
      case '-': cout << "Результат равен " << op1 - op2 << endl;
      case '/': if (op2 != 0.0)
                  cout << "Результат равен " << op1 / op2 << endl;
                else
                  cout << "Деление на ноль" << endl;
    default:  cout << "Недопустимая операция" << endl;
              exit(EXIT_FAILURE); }                       // провал
    exit(EXIT_SUCCESS);                                   // все OK
}
```

Литеральные метки в ветвях case — это не переменные. Они представляют собой константы этапа компиляции (здесь '+', '*' и т. д.). В случае оператора switch две метки case не могут быть одинаковыми (если только они не относятся к разным операторам switch).

При выполнении значение выражения switch (в данном случае переменная ch) сравнивается (сверху вниз) с литералами case. Если значение выражения совпадает с меткой, то выполнение продолжается с операторов, следующих за этой меткой до конца оператора switch.

Если ни один из литералов не соответствует значению выражения switch, то это не ошибка. В таком случае выполняются операторы, следующие за ключевым словом default. Метка default не обязательна. Обычно она является последней в операторе switch, но может помещаться и в середину оператора. Если она отсутствует и нет меток, соответствующих значению выражения switch, то все операторы в конструкции switch пропускаются и выполняется следующий оператор. Заметим, что операторы в case-ветвях switch не нужно заключать в фигурные скобки. Они не изменят порядок выполнения — все операторы обрабатываются последовательно.

```
Введите операнд, операцию и другой операнд: 22+2
Результат равен 24
Результат равен 44
Результат равен 20
Результат равен 11
Недопустимая операция
```

Рис. 4.23. *Результаты выполнения программы (некорректной) из листинга 4.27*

Результат выполнения программы из листинга 4.27 показан на рис. 4.23. Оператор switch не представляет собой конструкцию с большим числом ветвей. Это многовходовая конструкция. Если нужна разветвленная структура, ее можно построить из оператора switch, используя операторы break, goto или return, для завершения отдельной ветви. Листинг 4.28 показывает лучший вариант нашего оператора switch. Результат программы будет такой же, как на рис. 4.22.

Листинг 4.28. Калькулятор с оператором switch **(улучшенная программа)**

```
#include <iostream>
#include <cstdlib>
using namespace std;

void main(void)
{
  double op1, op2; char ch;
  cout << "Введите операнд, операцию и другой операнд: ";
  cin >> op1 >> ch >> op2;
```

```
    switch(ch) {                                          // обязательные фигурные скобки
      case '+': cout << "Результат равен " << op1 + op2 << endl;
                break;
      case '*': cout << "Результат равен " << op1 * op2 << endl;
                break;
      case '-': cout << "Результат равен " << op1 - op2 << endl;
                break;
      case '/': if (op2 != 0.0)
                  cout << "Результат равен " << op1 / op2 << endl;
                else
                  cout << "Деление на ноль" << endl;
                break;
    default:  cout << "Недопустимая операция" << endl;
              break; }                                    // здесь break необязателен
    exit(EXIT_SUCCESS);                                   // следующий оператор
}
```

Оператор break внутри switch передает управление следующему оператору после закрывающей скобки switch. Оператор exit() завершает функцию, как и ранее.

Для передачи управления из оператора switch можно использовать оператор goto, но в большинстве случаев достаточно оператора break. В некоторых программах break помещается даже перед закрывающей фигурной скобкой switch. Такой break бесполезен, но может предотвратить ошибки, если к оператору switch добавляются другие ветви, а break при этом не вставляется. Не столь важный момент, но всегда полезно облегчить работу тому, кто будет сопровождать программу.

Многовходовость оператора switch (а не наличие большого числа ветвей) можно использовать, чтобы избежать повтора кода, когда в нескольких ветвях требуется одна и та же обработка. Возьмем, например, переменную response, которая содержит ответ пользователя на запрос приложения. Предположим, нужно выполнять какое-то одно действие, когда пользователь вводит 'y' или 'Y', и другое, когда вводится 'n' или 'N'. И даже третье, когда получен иной ответ. Оператор switch для обработки ответа пользователя может выглядеть так:

```
switch (responce) {
  case 'y': case 'Y':
    cout << "Спасибо за подтверждение\n"; break;
  case 'n': case 'N':
    cout << "Запрос отменен\n"; break;
  default: cout << "Некорректный ответ\n"; }
```

Если, например, получен ответ 'y', то выполняется неявный пустой оператор между case 'y': и case 'Y':, а затем оператор за следующей меткой (в данном случае 'Y').

Конечно, то же самое можно сделать с помощью последовательности операторов условия, но оператор switch делает это лучше — программу легче читать, а за ее выполнением проще проследить. Очень мощное инструментальное средство.

Итоги

Здесь много рассказывалось об управляющих конструкциях языка C++. Этот язык содержит традиционные операторы условия и циклы, позволяющие программисту представлять сложные алгоритмы. Уникальная особенность C++ — возможность помещать логические выражения в операторы условия и циклы. В сочетании со способностью C++ интерпретировать любое ненулевое значение как `true` программист получает в свое распоряжение сильный инструмент для написания лаконичного и мощного кода.

Начинающим программистам иногда трудно понять такой код. При изучении C++ важно выделить время для освоения данных средств. Слишком часто программисты концентрируются на изучении классов и объектов, пренебрегая навыками, составляющими основу написания профессиональных программ на C++.

Другие управляющие конструкции C++, переходы и переключатели, не так необходимы для создания программ, как условные операторы и циклы. Можно написать на C++ вполне работоспособный код и без них. Но они неоценимы как инструмент вашего профессионального мастерства. Без применения данных операторов и их корректного использования ваш код нельзя будет считать профессиональной программой C++, так что не забудьте выделить время на изучение и практическое освоение этих элементов языка.

Глава 5

Агрегирование с помощью типов данных, определяемых программистом

Темы данной главы

- ✔ Массивы как однородные агрегаты
- ✔ Структуры как неоднородные агрегаты
- ✔ Объединения, перечисления и битовые поля
- ✔ Итоги

В предыдущей главе рассказывалось об инструментальных средствах для реализации алгоритмов в C++. С помощью условных операторов, переходов и циклов можно указать, как именно должны выполняться вычисления и в какой последовательности. В данной главе мы сделаем шаг вперед и научимся писать хорошо спроектированные программы C++. Здесь обсуждается расширение набора имеющихся в языке типов данных.

C++ позволяет программисту определять совокупности данных: массивы (однородные совокупности), структуры (неоднородные) и производные типы. Как уже говорилось, они иногда называются типами, определяемыми пользователем. Это точка зрения разработчика компилятора, а не программиста. Для программиста пользователь — тот, кто выполняет программу (или работает с ее результатами). Вот почему здесь типы данных, определяемые в программе, называются *типами, определяемыми программистом*.

В программе можно определять переменные типов (еще они называются объектами) точно так же, как и переменные встроенных типов (целые, символьные и пр.).

Для определения переменных в C++ используется тот же синтаксис. Правила работы с ними те же. Фактически программист может просто расширить набор типов данных C++, добавив собственные типы данных. Определяемые программистом типы данных добавляются к стандартным типам. Их можно использовать как блоки для определения новых, еще более сложных типов. Данная глава посвящена обсуждению массивов, структур и их вариаций: объединений, битовых полей и перечислений.

Классы C++ представляют собой совокупности данных и функций, но к ним можно будет перейти только после того, как мы подробнее обсудим функции C++. Описание функций уже содержалось в главе 2 — этого достаточно для понимания основных концепций функций, но не для понимания классов и способов их использования.

Здесь приведено больше базовой информации, а обсуждаемый материал разнообразнее и сложнее. Возможно, приступить к изучению классов будет проще, если сосредоточиться на массивах (только одномерных) и структурах (но не иерархических). Объединения и битовые поля — методы программирования, имеющие мало общего с классами. Это не значит, что они не важны. К данной главе можно вернуться, когда вам потребуется расширить навыки программирования.

Насчет перечислений сказать трудно. Формально для понимания классов они не нужны, но программисты часто используют перечисления для определения размеров компонентов класса. В главе 9 даны некоторые примеры перечислений. Они интуитивно понятны, но если потребуется более углубленное описание перечислений, лучше вернуться к этой главе и найти их здесь.

Массивы как однородные агрегаты

Массив — это набор элементов данных одного типа. Визуально можно представить массив как непрерывные ячейки в памяти. Все они одного размера и представляют компоненты одного типа. Можно определять массивы целых типов, типов с плавающей точкой и символов, а также массивы любого определяемого программистом типа (если этот тип известен в том месте программы, где определяется массив).

Массивы как векторы значений

Обычные переменные, о которых рассказывалось в главе 3, называют скалярами или элементарными (простыми) переменными.

Они характеризуются одним значением. Иногда желательно различать отдельные компоненты значения, например целую и дробную часть числа с плавающей точкой. Но в языке такое различие не поддерживается. Он интерпретирует переменные как единое целое, а не нечто, состоящее из компонентов. Вот почему эти переменные называются скалярными или элементарными. Чтобы выделить отдельные части из числа с плавающей точкой, нужно придумать код C++, который это делает. Он не слишком сложный (имеются библиотечные функции), но заранее определенного в языке способа нет.

```
fraction = x - floor(x);          // получить дробную часть x
```

Здесь `x` и `fraction` — числа с плавающей точкой, а `floor()` — функция, определенная в библиотечном заголовочном файле `math.h` (или `cmath`), она возвращает преобразованное в `double` максимальное целое, не превышающее аргумента. Сам язык интерпретирует значения встроенных типов как элементарные.

Массивы представляют собой векторы. Их состояние характеризуется набором значений, а не одним значением. К каждому значению — компоненту можно обращаться с помощью специальной записи C++ (операции индекса).

Массивы полезны, когда их элементы в программе обрабатываются одинаково. Вот почему массивы должны быть однородными. Тогда программа может перебирать элементы массива, выполняя с каждым из них одни и те же операции. Важно, чтобы элементы массива были одного типа. Это предотвращает проблемы, возникающие, когда операция применима к одному компоненту массива, но неприменима к другому.

Массивы — упорядоченные совокупности данных. Это означает, что каждый элемент массива имеет предыдущий и следующий элемент, хотя есть два очевидных исключения: первый элемент массива не имеет предыдущего, а последний — следующего. Массив имеет имя, но отдельные элементы массива имен не имеют. Программа обращается к ним по имени с добавлением индекса, определяющего позицию элемента в упорядоченном наборе.

Массивы конечны. Число элементов массива должно быть известно во время компиляции и не может изменяться при выполнении программы. Программисту нужно решить, сколько элементов массива должно в нем храниться, задать это при написании программы и придерживаться данного решения — хорошее оно или плохое.

Это серьезное ограничение. Если программист выделит для массива слишком много пространства, то оно будет использоваться зря и программе может не хватить памяти для других целей. Если же не выделить достаточно места, то программа при выполнении испортит содержимое памяти, что нередко ведет к ее аварийному завершению или некорректным результатам. Когда программист захочет изменить размер массива, ему потребуется отредактировать программу, заново скомпилировать и скомпоновать исходный код. Для небольшой программы это просто, но в случае сложной программы или программы, распространяемой среди тысяч пользователей, возникнет серьезная проблема.

Иногда размер массива уже точно известен заранее. Например, массив, состоящий из рабочих часов разных дней недели, содержит семь компонентов. То же относится к компонентам, содержащим число дней в месяце. Другие массивы могут представлять шахматную доску со всеми ее клетками. В большинстве случаев приходится искать разумный компромисс, выделяя больше элементов, чем может потребоваться, но не слишком много (например, двойное количество). Такое решение "разумно", поскольку поддерживается кодом, проверяющим переполнение и предпринимающим "разумные" действия, когда оно происходит. Для некоторых "разумность" может означать завершение программы, для других — завершение ввода с уведомлением пользователя.

Иногда позиция элемента в массиве имеет для приложения определенный смысл. Например, имя врача, дежурящего в больничной палате, привязано к дню недели. Когда меняется дата, это не обязательно означает переход к первому элементу, затем ко второму и т. д. Некоторые массивы могут вовсе не содержать допустимые данные. Когда массивы используются таким образом, нужно проработать способ, позволяющий программе различать элементы с допустимыми данными от других элементов. Подобные массивы называются разрешенными.

Многие массивы являются непрерывными и соответственно используются. Первый элемент данных записывается в первый элемент массива, следующий — во второй. Когда цикл обрабатывает допустимые элементы массива, для прекращения итераций можно использовать счетчик. Еще один способ реализации непрерывных массивов состоит в применении специального значения, добавляемого после последнего действительного элемента массива. Когда цикл обрабатывает допустимые элементы, он мог бы останавливаться, встретив в массиве такое специальное значение. Это специальное значение называется контрольным (аналогично контрольным значениям, завершающим ввод, в главе 4). Оно должно отличаться от значений, которые может содержать массив.

Определение массивов в C++

Как и любая переменная C++, переменная типа массива должна предварительно определяться и лишь затем использоваться. Определение массива связывает имя массива с элементами массива и числом элементов. Как и любое определение, определение массива приводит к выделению памяти во время выполнения, как и другие определения, оно заканчивается точкой с запятой. Массивы можно определять по одному на каждой строке или комбинировать на одной строке несколько определений, например:

```
int hours[7]; char grade[35]; double amount[20];
```

На этой строке определяются три массива: массив hours[] из 7 целочисленных компонентов, массив grade[] из 35 символьных компонентов и массив amount[] из 20 компонентов с плавающей точкой двойной точности. Обратите внимание на добавление к имени массива квадратных скобок. Такое обозначение в тексте говорит о том, что речь идет о векторе с несколькими значениями, а не о скаляре с одним значением.

В случае массивов разных типов (как в предыдущем примере) каждый массив нужно определять отдельно, завершая определение точкой с запятой. Для массивов одного типа можно определить несколько массивов через запятую, завершив последний точкой с запятой. Допускается комбинирование определений массивов и скалярных типов, если их типы совпадают, например:

```
int category[7], i, num, scores[35], n;
```

Некоторые программисты выбирают имена массивов, используя множественное число. Когда массив передается функции в качестве параметра, можно сказать, что функция получает не одно значение, а набор значений, например sum(scores). Другие предпочитают использовать имена в единственном числе. При ссылке на отдельные элементы массива по индексу, например category[i], естественнее именовать их именно в единственном числе. Вообще говоря, это не столь важно.

Размер массива должен быть известен во время компиляции, но он не обязательно задается литеральным значением. Можно использовать определенный в #define символьный литерал, целочисленную константу или целочисленное выражение любой сложности. Единственное требование состоит в том, чтобы это выражение вычислялось на этапе компиляции, а на не этапе выполнения. Например:

```
#define MAX_RATES 35          // размер массива определен как значение #define
int const NUM_ITEMS = 10;     // размер массива - константа
int rates [MAX_RATES]; double amount[2*NUM_ITEMS];
```

Массив может инициализироваться в определении аналогично любой другой переменной C++. Программист подставляет начальные значения — как при инициализации скалярных переменных. Эти начальные значения задаются в списке значений в фигурных скобках. Поскольку запятые являются здесь разделителями, а не завершают список, последнее инициализирующее значение запятой не имеет.

```
int hours[7] = { 8, 8, 12, 8, 4, 0, 0 };  // 7 значений
int side[5] = (40,35,41 } ;                // другие элементы массива - это нули
char option[2] = { 'Y', 'N', 'y', 'n' }; // синтаксическая ошибка
int week[52] = { , , 40, 48 };            // синтаксическая ошибка
```

Первое значение инициализирует первый компонент массива, второе — второй и т. д. Начальные значения должны иметь тип, совпадающий с типом массива; если типы разные, должно обеспечиваться преобразование между ними. Это те же преобразования смешанных типов и выражений, о которых уже рассказывалось в главе 3. (Например, допускается инициализация компонентов массива типа double с помощью целочисленных значений.)

В данных примерах подставляются значения для каждого компонента массива hours[]. Подставляемых значений может быть меньше числа элементов массива, как для массива side[]. Компоненты в этом случае инициализируются, начиная с первого, пока не исчерпается весь список. Компоненты, оставшиеся без значений, инициализируются нулем соответствующего типа. Не следует задавать инициализирующих значений больше, чем число элементов в массиве, как для массива options[]. Нельзя пропускать некоторые компоненты, используя запятые, как сделано для массива week[]. JCL (Job Control Language) допускает такой синтаксис, но C++ — не JCL.

Как и скалярную переменную, переменную-массив, определенную в одном файле, можно использовать в алгоритмах другого файла. Для этого в данном файле должна объявляться переменная с тем же именем. Основная разница между определением и объявлением массива в том, что в объявлении не задается размер массива. При объявлении массива для него не выделяется память. (Это задача определения массива.) Хотя в C++ объявления и определения похожи, программист должен различать их.

Например, в некоторых других файлах могут потребоваться значения компонентов массива hours[] или понадобится вычислять эти значения для присваивания. В этом файле массив hours[] можно определить так:

```
extern int hours[];          // объявление: память не выделяется
```

Чтобы это объявление было допустимым, исходное определение массива hours[] следует разместить вне любой функции как глобальную переменную.

Аналогично объявлениям скалярных переменных, объявления массива используются для присваивания массиву адреса в памяти. После этого программа в данном файле может обращаться к элементам массива hours[], как будто массив был определен в том же файле. Поскольку объявления массивов (как и любые другие объявления) не выделяют память, инициализации они не поддерживают.

C++ позволяет программисту использовать синтаксис объявления для определения массивов. Это делается, когда размер массива задается числом инициализаторов, а не явной константой этапа компиляции. Например:

```
double rates[] = { 1.0, 1.2, 1.4 };          // три элемента
```

Здесь выделяются и инициализируются три элемента массива, несмотря на запись в объявлении rates[]. Такое определение аналогично следующему:

```
double rates[3] = { 1.0, 1.2, 1.4 };          // явный счетчик
```

Преимущество первого определения в том, что оно короче — не нужно указывать размер массива. Однако в первом определении размер массива не задается константой, а такая константа может быть полезной в алгоритмах обработки массива.

Один из способов решения данной проблемы состоит в вычислении количества элементов массива с помощью операции sizeof (см. главу 3). Разделив размер массива на размер одного компонента, можно получить число элементов массива.

```
int num = sizeof(rates) / sizeof(double);
```

Обратите внимание на последовательность тем в обсуждении массивов C++. Она аналогична обсуждению средств определения других данных. Каждый раз рассказывается о смысле очередного средства C++ (переменных в главе 3, массивов в этой главе, затем структур, классов, составных и производных классов), речь идет о синтаксисе определений (и объявлений), а затем о вопросах инициализации. Эта последовательность не случайна. Инициализация имеет очень важное значение в C++, и мы будем изучать методы инициализации, соответствующие каждому виду использования памяти в C++.

Операции с массивами

За обсуждением инициализации неизбежно следует описание операций с массивами. Что можно с ними делать? C++ предлагает для этого достаточно ограниченный набор средств. Здесь нельзя присвоить одну переменную-массив другой, недопустимо их сравнивать, складывать, умножать и т. д. Единственное, что можно делать с массивами,— передавать их функции как аргумент. Таким образом, чтобы присвоить один массив другому, сравнить два массива и т. д., придется написать собственный код или использовать библиотеку функций.

Все операции можно выполнять только с отдельными элементами массива. При копировании одного массива в другой достаточно скопировать по отдельности каждый элемент массива. При сравнении массивов сравниваются соответствующие элементы. В этих операциях на отдельные элементы массива ссылаются с помощью операции индекса.

Например, side[2] обозначает элемент массива side с индексом 2. В любом случае side[2] — обычная скалярная целочисленная переменная. Поскольку side[] — это массив целых значений, с side[2] можно делать все то, что разрешается делать с целочисленной переменной как с l-значением или r-значением. Различие только в имени — вместо идентификатора целочисленной переменной используется имя массива, плюс индекс и операция индекса:

```
side[2] = 40;                    // используется как l-значение
num = side[2] * 2;               // используется как r-значение
```

На первой строке side[2] получает значение 40, которое сохраняется по соответствующему адресу. На второй — значение по адресу side[2] умножается на 2 и результат сохраняется в переменной num (она должна быть числовой). Как видно, отдельные элементы массива не имеют своих имен. Их имена составляются из имени массива и значения индекса.

В C++ используется вполне обычное обозначение элементов массива. Необычно лишь то, что C++ интерпретирует квадратные скобки как операцию, а не просто как обозначение элемента. Эта операция имеет высокий приоритет — она находится в верхней части таблицы операций C++ (см. таблицу 3.1). Как у любой операции, у нее есть операнды — имя массива и значение индекса. Эта операция применяется к имени side и значению 2, а результатом будет side[2] — имя компонента массива.

Сказанное звучит довольно абстрактно, и непонятно, как это применить на практике. Какая разница, операция это или специальное обозначение? Пока что различий не видно, но позднее данная операция будет использоваться в некоторых интересных контекстах.

Индекс не обязан быть литеральным значением и даже значением, известным на этапе компиляции. В качестве индекса может применяться любое числовое выражение. Если выражение имеет тип с плавающей точкой, символьный, short или long, оно преобразуется в целое. Здесь, например, на этапе выполнения вызывается функция foo(), а возвращаемое ею значение используется для вычисления индекса:

```
side[3*foo()] = 40;              // допустимо?
```

Чтобы это было законно, нужно определить функцию foo(), а возвращаемое ею значение (индекс) должно быть определено в диапазоне индексов. Если значения присвоены только части элементам массива, то данный индекс должен быть индексом одного из этих элементов. Когда значения присвоены всем элементам массива, индекс должен лежать в границах между первым и последним элементом. Индексы вне границ массива будут ссылаться на не принадлежащие ему ячейки памяти и, следовательно, они не должны рассматриваться как элементы массива.

Проверка допустимости индекса

А теперь — внимание... Программист не может произвольно выбирать диапазон значений индексов массивов. В C++ они фиксированы. Это неприятно, поскольку часто желательно придать индексу некий смысл. Например, хочется хранить в массиве revenue[] данные по обороту с 1997 по 2006 гг. Было бы удобно использовать данный диапазон как значения индексов. Другие языки позволяют программисту выбирать диапазоны индексов, но в C++ это не так.

Интересы разработчиков компилятора взяли верх над интересами прикладных программистов. В C++ диапазон индексов не только фиксирован. Индексы здесь начинаются с 0.

Да, индекс первого компонента любого массива C++ — 0, а не 1. Это очень важно.

Например, если массив `side[]` содержит пять компонентов, то допустимыми элементами массива будут `side[0]`, `side[1]`, `side[2]`, `side[3]` и `side[4]`. Обратите внимание, что `side[5]` не будет допустимым элементом массива.

Что случится, если по ошибке написать `side[-1]`, `side[5]` или `side[6]`? Сообщит ли компилятор о такой ошибке? Нет. Значением индекса может быть значение этапа выполнения, на момент компиляции не известное. Разработчики компилятора отказались от такой проверки. Даже если индекс при компиляции представляет собой литеральное значение, которое легко проверить, компилятор этого не делает. Допустимость индекса не проверяется — C++ предоставляет данную задачу программисту.

Если в программе написано `side[-1]`, то это, наверное, что-то значит, и в обязанности компилятора не входит разгадывать намерения программиста. Так что встроенная проверка допустимости индексов на этапе компиляции не предусмотрена.

А есть ли проверка на этапе выполнения? Ведь некоторые другие языки проверяют на допустимость каждую ссылку на компоненты массива. Но не C++. Такая проверка индекса или индексного выражения влияет на производительность, а в C++ это "священная корова". А если в вашей программе производительность не критична и желательно проверять допустимость индексов при выполнении программы — нет проблем. Сделайте это сами — сравните значения индексов с границами массива. Встроенной проверки индексов нет.

Вероятно, при разработке языка предполагалось, что программисту не требуется помощь со стороны компилятора и системы выполнения — он знает, что делает. Нет нужды говорить, что такое предположение безосновательно, и ошибки в индексах — частый источник проблем для программистов, работающих с C++.

Причина такой негибкости (унаследованной из C) в том, что в качестве адреса первого элемента массива используется имя массива. Смещение первого элемента от начала равно нулю. Смещение второго элемента равно длине элемента (она зависит от типа), смещение третьего — двум длинам. Компилятору известен размер элемента, и проще вычислить адрес элемента по смещению, а не по его позиции в массиве.

Когда значение индекса недопустимо, для вычисления адреса компонента массива в памяти компилятор все равно использует индекс как смещение. В результате программа портит свою память. Если данный адрес не используется в каких-то полезных целях, это может пройти незамеченным.



Осторожно! В C++ нет проверки допустимости индекса на этапе компиляции. Нет ее и на этапе выполнения. Содержимое памяти может быть запорчено программой. Будьте внимательны!

Рассмотрим некоторые последствия ошибок при обработке индексов. В листинге 5.1 показана программа, корректно присваивающая значения сторонам многоугольника, но некорректно их распечатывающая. Первое значение индекса равно 1, а последнее — 5. Результат выполнения программы показан на рис. 5.1.

```
40  41  42  43  13540
```

Рис. 5.1. *Вывод показывает ошибку в программе*

Листинг 5.1 Ошибки при переборе массива

```
#include <iostream>                          // или #include <iostream.h>
using namespace std;

int main()
{
  int size[5] = { 39, 40, 41, 42, 43 };
  for (int i = 1; i <= 5; i++)               // плохое начало, плохой конец
    cout << " " << size[i]; cout << endl;
  return 0;
}
```

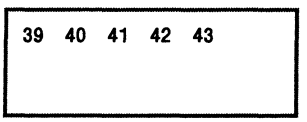


Рис. 5.2. *Правильный вывод показывает ошибку в обработке массива*

В данном случае проверка вывода показывает, что в программе ошибка. Но иногда, если программист упорно делает ошибки, проверка результата ошибки не показывает. В листинге 5.2 представлена программа, некорректно присваивающая стороны многоугольника и некорректно их выводящая. Она не использует принадлежащего массиву адреса side[0]. Вместо этого она работает с не принадлежащим массиву адресом side[5]. Вывод корректен, хотя программа портит ячейку памяти по адресу side[5].

Листинг 5.2. Ошибка, скрытая корректным выводом

```
#include <iostream>                                          // или #include <iostream.h>
using namespace std;

int main()
{
  int size[5];
  size[1] = 39; size[2] = 40; size[3] = 41; size[4] = 42; size[5] = 43;
  for (int i = 1; i <= 5; i++)                               // плохое начало, плохой конец
    cout << " " << size[i]; cout << endl;
  return 0;
}
```

Насколько опасна испорченная память? Если эта память в данной программе не выделяется под что-то полезное (а есть немало машин с большими объемами свободной памяти), то проблем не будет. Если же такая память используется программой, то ошибку найти довольно трудно. Как показано в листинге 5.2, трудно даже понять, что программа некорректна и где начинать искать ошибку. Из рис. 5.3 видно, что некорректно значение a[0]. Оно изменяется с 11 на 43 даже при отсутствии второго присваивания a[0]. На вашей машине данная программа может портить содержимое памяти по-другому. Как бы то ни было, эта невинная на вид программа некорректна.

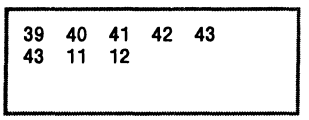


Рис. 5.3. *Массив a[] испорчен из-за обработки массива side[]*

Правильная итерация по компонентам массива должна начинаться с 0, а не с 1. Итерацию следует заканчивать на одно значение меньше, чем размер массива. Если размер равен 5, то корректной формой проверки будет i < 5; если размер равен 3, то корректная форма проверки i < 3. В общем случае, если число допустимых элементов массива хранится в переменной NUM, то корректной формой проверки на продолжение будет i < NUM. Перебор массива a[] в листинге 5.3 построен корректно. Заметим, что индекс определяется в первом цикле, а не в начале программы. Его имя известно только до конца функции. Следовательно, второй цикл не определяет эту переменную, а использует ее, как если бы она была

определена в начале функции. Мой компилятор (Microsoft Visual C++ 6.0) некорректно реализует стандарт C++: областью действия индексной переменной i должен быть только первый цикл, а не вся функция.

Листинг 5.3. Ошибка в одном месте портит содержимое памяти в другом

```
#include <iostream.h>
void main()
{ int a[3]; int size[5];
  a[1]=11; a[2]=12; a[3]=13;                                    // жертва порчи памяти
  size[1] = 39; size[2] = 40; size[3] = 41; size[4] = 42; size[5] = 43;
   for (int i = 1; i <= 5; i++)                                 // плохое начало, плохой конец
     cout << " " << size[i];
   cout << endl;
   for (i = 0; i < 3; i++)                                      // хорошее начало и конец
     cout << " " << a[i];
   cout << endl;
}
```

Программист все время должен думать о допустимости индексов в C++. При итерации по всем элементам массива ее следует начинать с индекса 0, а заканчивать, когда индекс на единицу меньше числа элементов массива. Это очень простое правило, его нетрудно запомнить. И в большинстве случаев оно соблюдается, но иногда каждый программист ошибается, обращаясь к элементам массива, и эти ошибки обходятся очень дорого, особенно, если они проявляются во время сопровождения. Если сложить все усилия, время и нервы, потраченные в индустрии ПО на ошибки обработки массивов, то результаты будут ужасающими. Вот почему программист, использующий C++, должен постоянно заботиться о допустимости индексов.



Советуем Начинайте итерацию по массиву со значения индекса, равного 0, и продолжайте ее, пока индекс не станет на единицу меньше числа элементов в массиве.

Многомерные массивы

C++ поддерживает многомерные массивы. Теоретически на число размерностей массива никаких ограничений не налагается. При определении многомерных массивов задается тип компонентов массива, имя массива, а потом, в отдельных квадратных скобках, число элементов в первом "измерении", втором, третьем и т. д. Например, двумерный массив целых чисел из двух строк и трех столбцов (пусть пока он будет простым) можно определить так:

```
int m[2][3];                    // 2 строки массивов по 3 элемента в каждом
```

Многомерные массивы могут инициализироваться с помощью синтаксиса, аналогичного инициализации одномерных массивов. Начальные значения перечислены в блоке с разделителями-запятыми:

```
int m[2][3] = { 10, 20, 30, 40, 50, 60 };
```

Здесь первые три значения попадают в первую строку матрицы, а последние три — во вторую строку. Для больших массивов можно указать группу значений, принадлежащих к одной строке, с помощью скобок, которые определяют область действия. Инициализаторы разделяются запятыми, это поможет сопровождающему приложение программисту идентифицировать данные в каждой строке:

```
int m[2][3] = { { 10, 20, 30 }, { 40, 50, 60 } };
```

Начальных значений, как и в одномерных массивах, может быть меньше, чем число элементов массива. Остальным элементам будет присваиваться нулевое значение. Например:

```
int m[2][3] = { { 10, 20 }, { 30, 40 } };
```

Это эквивалентно следующему явному определению, где первые три значения попадают в первую строку, а последние — во вторую:

```
int m[2][3] = { 10, 20, 0, 30, 40, 0 };
```

Подобно одномерным массивам, число начальных значений не должно превышать числа элементов в строке:

```
int m[2][3] = { { 10, 20, 30, 40 }, { 50, 60 } };          // ошибка
```

Метод определения размера массива путем спецификации начальных значений может использоваться и для многомерных массивов, но только частично. Разрешается опускать число строк, но нужно задавать число столбцов. Компилятор будет подсчитывать число начальных значений и определять число строк.

```
int m[][3] = { { 10, 20, 30 }, { 40, 50, 60 } };
```

Независимо от того, указывается число строк или нет, нельзя опускать число столбцов, это ошибка:

```
int m[2][] = { { 10, 20, 30 }, { 40, 50, 60 } };           // ошибка
```

Теоретически компилятор может подсчитывать группы строк и понимать структуру матрицы, однако он этого не делает. Возможно, лучше задавать размерность массивов явным образом.

Для доступа к элементам многомерного массива нужно несколько индексов — по одному на каждую размерность. Аналогично одномерным массивам, каждый индекс представляет смещение элемента, а следовательно, начинается с 0 и заканчивается значением, на единицу меньшим числа элементов массива. Например, первый элемент второй строки матрицы `m[][]` обозначается как `m[1][0]`. Это обозначение можно использовать как r-значение (операнд выражения), а также как l-значение (цель присваивания).

При переборе элементов многомерного массива следует использовать вложенные циклы. Во вложенных циклах многомерных массивов внутренний цикл завершает все свои операции сначала для одной итерации внешнего цикла, затем для следующей итерации внешнего цикла и т. д.

В листинге 5.4 показан вложенный цикл, который построчно выводит на экран каждый элемент матрицы `m[][]`. Внутренний цикл изменяет индекс `j` от 0 до 2 для каждого значения внешнего цикла `i` (который меняется от 0 до 1). Вывод программы показан на рис. 5.4.

Листинг 5.4. Пример операций с двумерным массивом

```
#include <iostream.h>
void main()
{ const int ROWS = 2, COLS = 3;
  int m[ROWS][COLS] = { { 10, 20, 30 }, { 40, 50, 60 } };
  for (int i=0; i < ROWS; i++)            // один раз для каждой строки
    { for (int j=0; j<COLS; j++)          // для каждого индекса i
        cout << " " << m[i][j];
      cout << endl; }                     // конец строки: один раз для каждого индекса i
}
```

Программисты, знакомые с другими языками, считают это обозначение многомерных массивов трудноватым (или, по крайней мере, новым). Иногда они ошибочно используют только один набор квадратных скобок и разделяют индексы запятыми. Например, вместо `m[1][0]` программист может использовать `m[1,0]`. К сожалению, компилятор в этом случае не сообщает об ошибке. Вместо этого он спокойно принимает данное выражение с разделителями-запятыми и предоставляет программисту самому разбираться в некорректности программы. На рис. 5.5 показан вывод программы из листинга 5.4, где `m[i][j]` ошибочно записано как `m[i,j]`.

```
10  20  30
40  50  60
```

Рис. 5.4. *Двумерный массив: построчный вывод*

```
0x34C4  0x34CA  0x34D0
0x34C4  0x34CA  0x34D0
```

Рис. 5.5. *Вывод программы из листинга 5.4 с `m[i][j]`, записанным как `m[i,j]`*

Причины здесь две, и обе унаследованы из языка C. Одна из них в том, что запятая — полноправная операция C++. Когда компилятор вычисляет разделенное запятыми индексное выражение `[i,j]`, он сначала вычисляет `i` (или 1), затем находит запятую, опускает значение `i` и вычисляет следующее выражение (т. е. 0). Затем компилятор возвращает данное значение в качестве индекса. Для различных целей `m[i][j]` можно записать как `m[j]`. Вторая причина в том, что `m[j]` с одним индексом — допустимое обозначение строки со смещением `j`. В многомерном массиве не обязательно указывать все индексы.



Внимание Для ссылки на компонент многомерного массива используйте форму с двумя наборами квадратных скобок: `a[i][j]`. Не применяйте разделитель-запятую, как в `a[i,j]`. Это ведет к проблемам.

Многомерные массивы поддерживаются в языке C++ как синтаксическое излишество — только для удобства программиста. "Внутри" они реализуются как одномерные массивы. Некоторые программисты предпочитают использовать одномерные массивы с компонентами `ROWS*COLS` и вычислять индекс элемента в `i`-той строке и `j`-той строке как `i*COLS+j`. (Не забывайте: индексы начинаются с 0 и заканчиваются значением `ROWS*COLS-1`). Листинг 5.5 показывает программу из листинга 5.4, где массив явным образом интерпретируется как одномерный. Вывод данной программы (см. рис. 5.5) будет таким же, как на рис. 5.4.

Листинг 5.5. Использование одномерного массива для реализации матрицы

```
#include <iostream>
using namespace std;

int main()
{
  const int ROWS = 2, COLS = 3;
  int m[ROWS * COLS] = { 10, 20, 30, 40, 50, 60 };  // тот же размер
  for (int i=0; i < ROWS; i++)
    { for (int j=0; j < COLS; j++)
        cout << " " << m[i*COLS + j];              // трудный способ
      cout << endl; }                              // конец строки (один раз для каждого i)
  return 0;
}
```

Какой способ обработки индексов лучше? Это зависит от приложения и личных предпочтений, поэтому рекомендации дать трудно.

Часто при выборе представления массива не важно, какой массив используется — одномерный или многомерный, один индекс или несколько. Программисту нужно лишь быть готовым к операциям с индексами.

Определение символьных массивов

Текст представляется в C++ в виде массивов символов (они часто называются *строками*), поэтому символьные массивы имеют здесь особое значение. Все, что говорится в данной главе о массивах (одно- и многомерных) применимо и к массивам символов.

Для операций с массивами (печать, сохранение в файл, копирование в другой массив, сравнение с другим массивом и т. д.) нужно знать, где массив заканчивается. Не всегда для этого доступно определение массива, а даже когда оно доступно, определение нередко бывает бесполезным, поскольку размер массива должен быть больше любого хранимого в нем набора элементов. Поэтому фактическое число элементов в массиве часто меньше, чем размер массива.

Как уже упоминалось выше, есть два подхода к решению этой проблемы. Один из них состоит в том, чтобы хранить в массиве счетчик элементов, а другой — в применении контрольного значения в конце массива. Для несимвольных массивов можно использовать любой метод. Для символьных массивов C++ применяет второй метод. При этом он использует нулевое контрольное значение, поскольку 0 — это специальный код, отличный от любого допустимого символьного кода. (Его часто называют *нулевым терминатором*, или просто *терминатором*.)

Чтобы отличить этот код от символа '0' (в коде ASCII это 48 в десятичном представлении и 0x30 в шестнадцатеричном), контрольное значение часто представляется как ESC-последовательность '\0' (0 в десятичном представлении и 0x0 в шестнадцатеричном).

Когда массив символов или литеральная строка передаются как аргумент любой библиотечной функции C++, она ожидает, что к тексту будет добавлено контрольное значение. При генерации функцией массива символов она добавляет такое значение в конец массива. В результате этот массив можно использовать в любой другой библиотечной функции:

```
char t[4] = { 'H','i','!','\0' );          // четыре элемента массива
cout << t << endl;                         // выводит "Hi!"
```

Здесь массив t[] инициализируется с помощью стандартного синтаксиса для одномерных массивов и передается функции-операции << как аргумент. Эта функция продолжает вывод строки символов, пока не встретит нулевой код. Тогда она прекращает работу.

В строковых литералах завершающий ESC-символ обязателен, но во многих контекстах можно использовать числовой 0. В следующем примере он применяется вместо ESC-символа '\0'. Некоторые программисты предпочитают символьную запись.

```
char t[4] = { 'H', 'i', '!', 0 };          // некоторые предпочитают '\0'
```

Такая запись не совсем обычна для инициализации символьных массивов большой длины. На этот случай C++ предусматривает специальное решение: вместо набора символьных значений можно использовать символьный литерал. Компилятор поймет, что вы имеете в виду, поместит каждый символ в соответствующую позицию массива и добавит в конец терминатор:

```
char t[4] = 'Hi!';                         // t[0] - это 'H', t[1] - это 'i', и т.д.
char u[] = "Сегодня прекрасный день"       // 24 символа с нулевым завершителем
```

Строковый литерал "Hi!" содержит 4 символа. Четвертый — это код 0. Строковый литерал "Сегодня прекрасный день" содержит 24 символа, включая код 0. Следовательно, в массиве u[] не 23 компонентов, а 24. Для хранения контрольного символа нужен дополнительный элемент массива. Если не зарезервировать для него место, возникнет проблема. Например, такая запись даст синтаксическую ошибку:

```
char v[3] = "Hi!";                         // Четыре начальных значения для трех элементов
```

Можно определить массив символов, где зарезервированного места больше, чем начальных символов, а можно определить такой массив и оставить его содержимое неопределенным:

```
char last[30]="Jones", first[30];          // имеется место
```

Доступ к обычным строковым компонентам аналогичен обычному массиву. Каждый компонент массива имеет тип char. Первый индекс равен 0.

```
t[0] = 'N'; t[1] = 'o';                    // t[] содержит "No!", а не "Hi!"
```

При работе с символьными и строковыми литералами важно помнить, когда требуются одиночные кавычки, а когда — двойные. Например, выше 'o' — это символьный литерал, а "o" — строковый литерал. Он состоит из двух символов — символа 'o' и символа '\0'.

Операции с символьными массивами

Отдельные символы можно присваивать один другому или сравнивать друг с другом. Их можно сдвигать, складывать и т. д. Для строк символов (массивов) ни одна из этих операций недоступна. К счастью, библиотека C++ поставляется с большим числом функций для работы со строками символов. В аргументах функций имя массива используется без индекса.

Функция strcpy() реализует присваивание массивов символов. Она воспринимает два аргумента-массива и копирует компоненты второго аргумента в соответствующие компоненты первого. Нулевой терминатор также копируется. В результате получается нормально сформированный целевой массив, который можно использовать в аргументах других функций:

```
strcpy(u,t);               // Теперь u[] также содержит "No!"
```

Поскольку ни одна функция не просматривает содержимое строки за пределами контрольного значения, очищать остаток строки нет необходимости. Перед вызовом этой функции строка u[] содержала "Сегодня прекрасный день\0", где "\0" — контрольный символ. После вызова функции она будет содержать "No!\0дня прекрасный день\0", но все, что содержится после "\0", теперь уже нерелевантно. (Здесь использование ESC-символа обязательно.) Для любых целей содержимым данной строки будет просто "No!".

С передачей строкового литерала в виде аргумента функции, ожидающей символьный массив, нет никаких проблем. Единственное требование состоит в том, чтобы функция не изменяла состояния массива, так как литерал — это константа и функцией изменяться не может.

```
strcpy(t, "Yes");          // теперь t[] содержит "Yes", плюс ноль
strcpy("Yes", t);          // этого делать нельзя - синтаксическая ошибка
```

Функция strcat() реализует для массивов символов операцию +=. Она также воспринимает два символьных массива как аргумент и копирует второй аргумент в первый. В отличие от strcpy() она не заменяет содержимое первого параметра новым содержимым, а просто добавляет его к текущему. Результатом будет конкатенация двух строк:

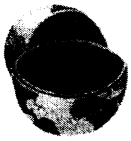
```
strcat(u, " means No!");          // u[] содержит "No! means No!"
```

Добавление символов начинается с позиции, где раньше был нулевой терминатор. Этот терминатор и то, что может находиться за ним, затирается — туда помещается новое содержимое. Нулевой терминатор вставляется после добавленных символов. В результате в массиве будет "No! means No!\0ный день\0". Поскольку функции C++ не просматривают то, что находится за терминатором, для всех целей это будет просто строка "No! means No!".

Функция strcmp() реализует операцию сравнения двух символьных элементов-массивов. Она поочередно сравнивает соответствующие символы, пока не найдет в одной позиции два разных символа или не достигнет завершающего строку контрольного значения. Если функция находит два разных символа, она сравнивает их в лексиграфическом порядке, т. е. определяет, какой из них следует первым в коде ASCII. Когда строки упорядочены и символ в первом аргументе предшествует символу во втором аргументе, strcmp() возвращает –1. Если строки не упорядочены и символ во втором аргументе предшествует символу в первом аргументе, strcmp() возвращает 1. При достижении контрольных значений в одной позиции она возвращает 0. Строки считаются в этом случае совпадающими.

Например, strcmp("Hi", "Hello") вернет 1. Строки не упорядочены. С другой стороны, strcmp("Handler", "Hello") вернет –1, как и strcmp("Hell", "Hello"), поскольку контрольное значение в первой строке сравнивается с символом 'o' во второй. Так как в коде ASCII символы в нижнем регистре следуют за символами в верхнем регистре, strcmp("hello", "Hello") возвращает 1 и выполняет ветвь true следующего оператора:

```
if (strcmp("hello", "Hello")) cout << "Не упорядочены\n";
```



Внимание Все библиотечные функции C++ прекращают обработку строки, когда находят завершающий ноль. Символы, следующие за нулем, для библиотечных функций недоступны. Для подсчета числа символов, предшествующих завершающему нулю, используйте функцию strlen().

Еще одна полезная библиотечная функция — strlen(), воспринимающая символьный массив в качестве аргумента и возвращающая число символов в строке (*предшествующих* нулевому терминатору). Например, strlen("Hello") возвращает 5, а strlen(t) — 3 (t содержит "Hi!"). Все библиотечные функции прекращают просмотр строки, как только встречают контрольный символ.

Строковые функции и порча содержимого памяти

Ни одна из этих функций не проверяет, хватит ли места для операции. Формально причина в том, что, получая аргумент, функция C++ не знает точно числа элементов в массиве, будь то символьный массив или массив иного типа. В действительности такая проверка может повлиять на производительность программы, а потому данная обязанность (наряду с другими вопросами) возлагается на программиста. Он все время должен думать о том, достаточно ли места для операции.

Кроме того, если два массива перекрываются в памяти, функции дают "неопределенные результаты", т. е. возвращаемые функцией результаты могут быть некорректными. В любом случае никакой гарантии нет.

В листинге 5.6 приведена программа, непрерывно работающая с доступной областью памяти. Что может быть проще считывания двух элементов и отображения их на экране? Программа передает функции >> массивы first[] и last[], предварительно заполнив их вводимыми символами и добавив нулевые терминаторы.

```
Введите имя: John
Введите фамилию: Johnson
n Johnson
```

Рис. 5.6. *Переполнение массива при вводе приводит к порче данных*

Вывод программы показан на рис. 5.6. Если данных достаточно мало, проблем не будет. Если данные длиннее, то проблема (в этом тривиальном примере) становится очевидной. Вряд ли можно считать, что 6 символов для имени и фамилии достаточно. Более того, доступны только 5 знакомест, так как шестое занимает терминатор. А 20 символов достаточно? Моя подруга Галина Белосельская-Белозерская столкнулась бы с проблемой — в ее имени-фамилии 32 символа, включая пробел и завершающий 0.

А вдруг на клавиатуре запала клавиша? Пользователь заметит ошибку и исправит запись, но другие ячейки памяти могут быть затерты без каких-либо на то указаний.

Листинг 5.6. Простой пример переполнения массива

```
#include <iostream>
using namespace std;

int main()
{
  char first[6], last[6];              // не слишком ли коротки эти массивы?
  cout << "Введите имя: ";             // я ввел "John\0" (5 символов)
  cin >> first;                        // нет защиты от переполнения
  cout << "Введите фамилию: "          // я ввел "Johnson\0" (8 символов)
  cin >> last;                         // нет защиты от переполнения
  cout << first << " " last << endl;   // просто для проверки результатов
  return 0;
}
```

Здесь first[] содержит строку "John\0" (последние два символа '\0' представляют ячейку памяти с нулевым содержимым). Массив last[] содержит "Johnson\0" (8 символов с завершающим нулем). Между тем, массив last[] содержит только 6 символов. Куда денутся два символа "n\0"? На моей машине массив first[] фактически следует в памяти за массивом last[]. Почему это так — не важно. Но очевидно, два лишних символа куда-то попадут. После ввода фамилии массив first[] содержит два символа "n\0", плюс то, что осталось от "John\0", т. е. "n\0hn\0". При попытке вывести содержимое first[] с помощью cout вывод прекращается после первого '\0', т. е. сразу после 'n'.

Это очень интересно и довольно опасно. На вашей машине программа может работать несколько по-другому. Некоторые компиляторы выделяют место фрагментами по 4 байта, так что массивы будут, фактически, содержать 8 символов, а потому для демонстрации порчи содержимого памяти потребуется более длинные имя/фамилия. Другие компиляторы не размещают массив first[] после массива last[]. Как бы то ни было, важно, что при обработке строк возможна порча содержимого памяти.

Хорошим решением проблемы является динамическое распределение памяти, но пока у нас нет необходимых для этого инструментальных средств. Другое практическое решение — ограничение числа помещаемых в массив символов. Функция ввода get() позволяет программисту задать ограничение на число считываемых символов:

```
cin.get(first,6);              // считывает до 5 символов + ноль
```

Если функция get() обнаруживает символ новой строки до того, как количество символов станет равным заданному ограничению минус один, ввод прекращается и символ новой строки '\n' остается во входном буфере в качестве первого символа для последующего считывания.

Если пользователь продолжает набирать текст, не нажимая клавишу Enter, ввод прекращается, когда количество считанных символов достигнет заданного ограничения минус один. После этого добавляется пустой завершающий символ для нормального формирования строки. Когда пользователь в конце концов нажмет клавишу Enter, лишние символы с символом новой строки останутся во входном буфере. В дальнейшем они будут считаны следующим оператором ввода (при его наличии).

Использование функции get() порождает две проблемы. Допустим, имя содержит только три символа (например, "Amy"). Далее вводится фамилия:

```
cin get(last, 6) // ввод прекращается по концу строки
```

Первое, что находит данный оператор в буфере ввода — символ новой строки, оставшийся от предыдущего вызова функции get() для чтения "Amy", поэтому он считывает символ новой строки и завершает работу. В результате в массиве last[] оказывается пустая строка. Уловили? То, что набирает пользователь, не попадает в массив last[] и остается в буфере ввода. Если первый ввод окажется слишком длинным ("Владимир"), то в first[] помещаются только пять символов с завершающим нулем. Остаток ввода ("мир") будет находиться в буфере, ожидая следующего запроса ввода. Что бы пользователь ни набрал в фамилии, это не попадет в массив last[]. Программа считает символы из буфера ввода ("мир") и остановится, встретив в буфере символ новой строки (он там остается).

Как видно, одна лишь функция get() не в состоянии выполнить задачу. Нужна функция ignore(), считывающая символы и отбрасывающая их. Для нее необходимо задать верхний предел — число отбрасываемых символов и ограничитель, который прекращает отбрасывание символов (здесь используется символ новой строки).

```
Введите имя: John
Введите фамилию: Smith
John Smith
Заказчик: Smith, John
John n
```

Рис. 5.7. *Переполнение массива при конкатенации приводит к порче других данных*

Данное решение проблемы переполнения ввода показано в листинге 5.7. Эта программа демонстрирует также следующую проблему, на сей раз относящуюся к копированию и конкатенации. Она формирует имя заказчика из фамилии, запятой, пробела и имени. Если число скопированных в массив name[] символов превышает размер массива, то символы все равно копируются, даже если они попадают в смежную с массивом память. Результаты выполнения показаны на рис. 5.7. Хотя массив name[] содержит "корректные" данные, массив last[] будет запорчен без всяких предупреждений, несмотря на то, что в программе нет никаких операторов, явно изменяющих его содержимое.

Листинг 5.7. Пример переполнения массива при конкатенации

```
#include <iostream>                          // или #include <iostream.h>
#include <cstring>                           // или #include <string.h>
using namespace std;

int main()
{
  char first[6], last[6]; char name[10];     // name = имя, фамилия
  cout << "Введите имя: ";
  cin.get(first,6); cin.ignore(2000,'\n');   // для удаления CR
  cout << "Введите фамилию: ";
  cin.get(last,6); cin.ignore(2000,'\n');    // останавливается на первом CR
  cout << first << " " << last << endl;
  strcpy(name,last);                         // копирует last[] в name[]
  strcat(name,", ");                         // добавляет запятую и пробел
  strcat(name,first);                        // добавляет first[] к name[]
  cout << "Заказчик: " << name << endl;
  cout << first << " " << last < endl;       // просто для данного примера
  return 0;
}
```

Здесь массив first[] содержит "John", а массив last[] — "Smith". Эти массивы защищены от переполнения при вводе. Затем выполняется конкатенация компонентов имени в массиве name[], например "Smith, John". Данная строка

содержит 12 символов, включая завершающий 0. Так как в массиве name[] только 10 символов, последние два символа ("n\0") попадают в другое место. На моей машине они оказались в массиве last[]. В результате массив last[] содержит эти два символа и то, что осталось от "Smith", т. е. "n\0itn\0". Когда последний оператор программы выводит имя, оно оказывается корректным, а при выводе фамилии появляется только "n".

Чтобы устранить проблему, C++ предлагает библиотечные функции strncpy() и strncat() из заголовочного файла string.h. Они аналогичны функциям strcpy() и strcat(), но имеют третий аргумент, ограничивающий число копируемых символов.

Программа в листинге 5.8 показывает их использование. Функция strncat() завершает копирование при получении заданного числа символов (или достижении конца второго аргумента) и добавляет нулевой терминатор. Все надежно. Функция strncpy() добавляет нулевой терминатор, только когда длина второй строки меньше указанного предела. Если достигается предел, она завершает копирование без добавления контрольного символа. Следовательно, strncpy() не всегда создает правильно сформированную строку. Использовать ее небезопасно. Рис. 5.8 демонстрирует, что применение функции strncat() не защищает целевой массив от переполнения. В массиве name[] содержатся усеченные данные ("Smith, Joh" вместо "Smith, John"), но они сформированы правильно.

```
Введите имя: John
Введите фамилию: Smith
John Smith
После копирования: Smith
Заказчик: Smith, Joh
John
```

Рис. 5.8. *Усечение данных предотвращает порчу содержимого памяти*

Листинг 5.8. Пример предотвращения переполнения массива при конкатенации строк

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{
  char first[6], last[6]; name[10];
  cout << "Введите имя: ";
  cin.get(last,6); cin.ignore(200,'\n');          // для удаления CR
  cout << "Введите фамилию: ";
  cin.get(last,6); cin.ignore(200,'\n');          // останавливается на первом CR
  cout << first << " " << last << endl;
// strncpy(name, last, 4);                        // нет нуля, если длина>=счетчика
  strcpy(name,last);
  cout << "После копирования: " << name << endl;
  strcat(name ", ");
  strncat(name,first,3);
  cout << "Заказчик: " << name << endl;
  cout << first << " " << last < endl;            // просто для данного примера
  return 0;
}
```

Похоже, принцип усечения выбран неверно. Массив name[] содержит 10 символов, а строка "Smith, Joh" — 11 символов, включая нулевой терминатор. Куда денется этот 0? На моей машине он оказывается первым символом в массиве last[]. Вместо "Smith" там оказывается "\0mith". Когда последний оператор cout выводит массив last[], он находит там нулевой символ и завершает работу, ничего не отображая.



Советуем При любых операциях со строками нужно убедиться, что вводимые данные не переполняют символьный массивов и не портят не относящихся к программе данных. Такие ошибки очень трудно найти, так как не всегда обнаруживается порча данных.

Звучит пугающе? Так и было задумано. Программисты, применяющие C++,— фигуры влиятельные. Они пишут программы, затрагивающие интересы многих. Сам язык — великий и могучий. Но он опасен в неопытных руках, подобно револьверу или автомобилю. Поэтому нужно сделать все возможное, чтобы правильно использовать его потенциал.

Двумерные символьные массивы

Символьные массивы могут иметь несколько измерений. Для многих задач обработки текста удобно организовать массив в виде строк и столбцов, создать "массив массивов" — двумерный массив типа char. Рассмотрим, например, массив из дней недели. Требуется определить, содержат ли введенные данные в массиве day[ ] слово "Sunday", "Monday", "Tuesday" и т. д. (воскресенье, понедельник, вторник...). Хотя это слова разной длины, мы представим их как двумерный массив символов из семи строк (для 7 дней недели). Строки должны быть одной длины. Максимальное название ("Wednesday") — 9 символов. Учитывая завершающий символ, это будет 10 столбцов.

```
char days[7][10], day[10];
```

Если нужно проверить, содержит ли массив day[ ] допустимый день недели, достаточно сравнить этот массив с каждой строкой массива days[][]. Можно сделать это в лоб, сопоставляя каждый элемент в i-той строке (i меняется от 0 до 6). Если символы в массиве day[] те же, что в i-той строке массива days[][], то итерация заканчивается на индексе i, так как введенные данные найдены в массиве. Следовательно, во внешнем цикле нужно знать, что происходит во внутреннем цикле. Обычно для этого используется управляющий флаг (здесь переменная found). Перед началом внутреннего цикла она устанавливается в 1 (true). Если внутренний цикл обнаруживает разные символы, то данная переменная устанавливается в 0 (false), показывая, что слово не найдено. Если все символы в массиве day[] и в i-той строке массива days[][] совпадают, то присваивание found = 0 не выполняется — флаг остается равным 1, оператор break завершает внешний цикл:

```
for (i = 0; i < NUM; i++)
{ found = 1; j = 0;
  do {
     if (days[j] != days[i][j])          // слово не найдено
        { found = 0; break; }            // стоп, сделать для следующего i
          j++;
          } while (day[j] != '\0');
  if (found == 1) break; }               // прервать внешний цикл
```

Некоторые программисты, знакомые с C++, могут написать последние строки более кратко:

```
do
{ if (day[j] != days[i][j++])          // сравнение и инкремент
  { found = 0; break; }                // стоп, сделать для следующего i
   } while (day[j] != '\0');           // отдельного j++ не требуется
  if (found) break; }                  // любое ненулевое значение - true
```

Применение такой формы оператора условия будет вполне безопасным. Однако комбинировать сравнение и инкремент безрассудно. Здесь j используется в двух подвыражениях, а в главе 3 говорилось, что неизвестно, в каком порядке они вычисляются. На моей машине данный код выполняется некорректно. На вашей может быть иначе. Как бы то ни было, предусмотрительный программист должен знать об этом и избегать подобного стиля программирования.

Часто обработку такого рода упрощают с помощью библиотечной функции strcmp(). В листинге 5.9 показана программа, запрашивающая у пользователя день недели, выполняющая поиск в двумерном массиве символов и выводящая на экран номер найденного дня. Обратите внимание, что после завершения цикла поиска программа должна определить, почему это произошло: найдено слово или кончились элементы массива, а совпадение не обнаружено. Есть несколько способов реализовать это. Здесь проверяется индекс i. Если он равен числу элементов массива, значит поиск был безуспешным. Если он меньше числа элементов, следовательно, поиск завершился по оператору break. На рис. 5.9 показаны результаты выполнения. Программа не тестировалась на все возможные вводимые значения, а это непредусмотрительно: в инициализации массива слово "Friday" набрано с опечаткой.

```
Введите день недели или 'end' для завершения ввода: Sunday
Вы ввели: Sunday
Sunday имеет номер 1
Введите день недели или 'end' для завершения ввода: Thursday
Вы ввели: Thursday
Thursday имеет номер 5
Введите день недели или 'end' для завершения ввода: saturday
Вы ввели: saturday

Ввод "saturday" некорректен

Введите день недели или 'end' для завершения ввода: end
Вы ввели: end
Спасибо, что воспользовались программой
```

Рис. 5.9. *Неполное тестирование не выявляет ошибок*

Листинг 5.9. Применение двумерного массива символов для поиска

```
#include <iostream>
#include <cstring>
using namespace std;
#define NUM 7                                   // вдруг число дней в неделе изменится...

int main()
{
  int i; char day[10];
  char days[NUM][10] = { "Sunday", "Monday", "Tuesday",
       "Wednesday", "Thursday", "friday", "Saturday" };
  do {                                          // пока пользователь не введет "end"
    cout << "Введите день недели или 'end' для завершения ввода: ";
    cin.get(day,10); cin.ingore(2000, '\n');    // непредусмотрительно
    cout << "Вы ввели: " << day << endl;
    if (strcmp(day,"end")==0) break;
      for (i = 0; i < NUM; i++)
        if (strcmp(day,days[i][==0) break;      // остановить, если найдено
      if (i==NUM)                               // проверить, почему мы здесь
      cout << "Ввод \"" << day << "\" некорректен\n";
    else
      cout << day << " имеет номер " << i+1 << endl;
  } while (1==1);
  cout << "Спасибо, что воспользовались программой" << endl;
  return 0;
}
```

Переполнение массива в алгоритме вставки

Приведенный в листинге 5.9 алгоритм использует массив данных (дни недели) только для поиска, поэтому вопрос переполнения массива здесь не возникает. Размер массива совпадает с числом его элементов. Обычно массивы заполняются данными до того, как они используются для последующей обработки. Данные загружаются в начало массива, и каждый следующий элемент добавляется после текущего последнего элемента. В каждый момент времени часть массива, занятая данными, представляет собой непрерывный набор элементов, а вторая, свободная часть массива — доступные адреса, но она не содержит действительных данных.

Если массив подвергается обработке, то операции производятся не с каждым элементом массива, а с каждым элементом его непрерывной части, заполненной данными. Таким образом, часто необходимо отслеживать, сколько действительных элементов данных находится в массиве. В алгоритме вставки элементов массива нужно побеспокоиться о контроле переполнения.

В листинге 5.10 показан алгоритм для занесения данных транзакций. Это расширение примера, представленного в главе 4. В прежних примерах для анализа завершения конца ввода использовалось фиктивное отрицательное значение. Здесь это делается более цивилизованным путем — пользователя просят набрать "end" (конец). (Простое нажатие клавиши Enter — не очень хорошая идея, ведь ее можно нажать случайно.) Чтобы можно было обрабатывать вводимые данные как текст и числа, программа помещает их в массив `buff[]` и проверяет наличие контрольного значения "end". Если его там нет, то она преобразует строку в число с плавающей точкой при помощи библиотечной функции `atof()`, определенной в файле `cstdlib` (или `stdlib.h`). В имени функции `'a'` означает ASCII, `'to'` — "в", а `'f'` — ... можно было бы сказать `float`, но на самом деле функция возвращает значение `double`. Существуют также функции `atoi()` (ASCII в `int`), `atol()` (ASCII в `long`), но функции `atod()` нет. Эти функции могут преобразовывать до 100 символов, поэтому с 20-символьным буфером они справятся. Если буфер не содержит числовых данных, `atof()` возвращает 0 и программа выводит предупреждение. Неплохо проверить, содержат ли вводимые данные что-либо, кроме числа, но это потребовало бы применения функций `strtod()` и `strtol()` (строка в `double` и в `long`), а также использования указателей, к чему мы еще не готовы.

В противном случае программа накапливает значение `total`, сохраняет введенные данные в массиве и увеличивает индекс. При завершении ввода переменная `count` будет содержать число допустимых значений в массиве. Вот почему в дальнейшей обработке (печать значений транзакций) используется цикл с итерацией до достижения индексом значения `count`, а не числа элементов массива `NUM`. Обратите внимание на применение библиотечной функции `width()`, которая задает минимальное число позиций вывода, выделяемых для следующего значения. По умолчанию это 0, т. е. значение занимает столько позиций, сколько для него требуется. Если число символов в выводимом значении меньше, чем "ширина" (количество знакомест), то остальная часть заполняется пробелами (числа выравниваются вправо, строки — влево). Если число символов превышает ширину `width()`, то ширина игнорируется, и значение займет столько позиций, сколько необходимо.

```
Введите сумму (или 'end' для завершения): 22
Введите сумму (или 'end' для завершения): 33
Введите сумму (или 'end' для завершения): 44
Введите сумму (или 'end' для завершения): 55
Введите сумму (или 'end' для завершения): end

Общая сумма 4 значений равна 154

Ном. опер.   Сумма

   1          22
   2          33
   3          44
   4          55
```

Рис. 5.10. *Использование непрерывного массива для сохранения вводимых данных*

Присутствующие в программе отладочные операторы выводят на экран вводимые данные и данные, преобразованные в числовую форму. Ошибки нередко происходят при некорректной интерпретации в программе данных, но они остаются незамеченными, поэтому полезно проверять, как программа воспринимает ввод. Пример ее выполнения показан на рис. 5.10.

Программа из листинга 5.10 делает полезную работу, используя для сохранения данных непрерывный набор элементов массива. Второй цикл не выводит все доступные элементы массива, а отображает только помещенные в массив в первом цикле. Между тем программа не предотвращает переполнение, если число вводимых значений превысит длину массива.

Листинг 5.10. Заполнение непрерывного массива данными транзакций

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

int main ()
{
  const int NUM = 100;                                  // конечно, размер может изменяться
  double total, amount, data[NUM]; int count;
  char buff[20];
  total = 0.0; count = 0;                               // инициализация текущих данных
do {                                                    // пока пользователь не введет "end"
  cout << "Введите сумму (или 'end' для завершения): ";
  cin/get(buff,20); cin.ignore(2000,'\n');
// cout << "Вы ввели '" << buff << "'" << endl;         // отладка
  if (strcmp(buff,"end")==0) break;
  amount = atof(buff);                                  // преобразует в double до 100 знаков
// cout << "Сумма операции: " << amount << endl;        // отладка
  if (amount <= 0)
    cout  << "Это значение отбрасывается как некорректное.\n";
          << "Повторите ввод.\n";
  else
    { total += amount;                                  // обработка текущих данных
      data[count] = amount;
      count++; }
    } while (1 == 1);
  cout << "\nОбщая сумма " << count << " значений равна "
       << total << endl;
  if (count == 0) return 0;
    cout << "\nНом. опер. Сумма\n\n";
    for (int i = 0; i < count; i++)
      { cout.width(4); cout << i+1;
        cout.width(11); cout << data[i] << endl; }
  return 0;
}
```

Чтобы избежать переполнения массива и порчи памяти, первый цикл должен проверять, указывает ли индекс count на допустимый элемент массива. Не забывайте, что первым недопустимым значением индекса будет NUM, число элементов массива, поэтому, пока count меньше NUM, значение можно сохранять в массиве. В противном случае ввод нужно прекратить.

Реальные программы содержат длинные массивы, для проверки на переполнение придется вводить сотни или тысячи значений. Это требует много времени и не исключает ошибок. Версия программы с защитой от переполнения показана в листинге 5.11. Чтобы избежать работы с большими наборами данных, размер массива уменьшен до 3 — полезный метод отладки.

Листинг 5.11. Ввод данных в массив с защитой от переполнения

```
#include <iostream>
#include <cstring>
#include <cstdlib>                          // для поддержки atof()
using namespace std;

int main ()
// { const int NUM = 100;                   // размер может изменяться
{
  const int NUM = 3;                        // только для отладки
  const chat LAST{} = "end";                // литерал для завершения
  double amount, total, data[NUM];
  char buff[20]; int count;
  total = 0.0; count = 0;                   // инициализация текущих данных
do {                                        // пока пользователь не введет "end"
  cout << "Введите сумму (или '" << LAST <<"' для завершения): ";
  cin/get(buff,20); cin.ignore(2000,'\n');
  if (strcmp(buff,LAST)==0) break;          // конец ввода данных
  amount = atof(buff);                      // преобразует в double до 100 знаков
  if (amount <= 0)
    cout  << "Это значение отбрасывается как некорректное.\n"
          << "Повторите ввод.\n";
  else if (count < NUM)
  { total += amount;                        // обработка текущих данных
   data[count] = amount;
   count++; }
  else
  { cout << "Не хватает памяти: ввод прекращен\n";
   break; }
  } while (1==1);
  if (strcmp(buff,"end") != 0)
    cout << "Значение " << amount << " не сохранено\n";
   cout  << "\nОбщая сумма " << count << " значений равна "
         << total << endl;
  if (count == 0) return 0;
  cout << "\nНом. опер. Сумма\n\n";
  for (int i = 0; i < count; i++)
    { cout.widht(4); cout << i+1;
      cout.width(11); cout << data[i] << endl; }
  return 0;
}
```

В примере из предыдущего раздела массив символов не изменялся и значение индекса было надежным индикатором причины завершения цикла. Если оно меньше числа элементов массива, это означало, что элемент найден. В противном случае цикл безуспешно проверил все элементы. В данном примере подобный подход не будет надежным на 100%, так как count может достигать значения NUM, когда пользователь ввел в точности NUM значений. Для больших массивов это очень редкий случай, поэтому некоторые программисты не принимают его в расчет, что неверно. В листинге 5.11 проверяется, ввел ли пользователь "end". Если нет, то цикл был завершен по причине переполнения массива. Результат выполнения показан на рис. 5.11.

```
Введите сумму (или 'end' для завершения): 22
Введите сумму (или 'end' для завершения): 33
Введите сумму (или 'end' для завершения): 44
Введите сумму (или 'end' для завершения): 55
Не хватает памяти: ввод прекращен
Значение 55 не сохранено

Общая сумма 3 значений равна 99

Ном. опер.   Сумма

   1          22
   2          33
   3          44
```

Рис. 5.11. *Использование короткого массива для проверки защиты от переполнения*

Эта версия программы демонстрирует также использование ключевого слова const с массивом LAST[], что позволяет сопровождающему приложение программисту легко заменить терминатор на "finish", пустую строку или на что-либо еще, не выискивая в программе все вхождения "end". Это же относится к символьному литералу NUM. Заменить константу в одном месте намного легче, чем глобально править всю программу. Однажды, когда мне понадобилось изменить размер массива со 100 на 300, я воспользовался командой replace. Это было финансовое приложение, где учитывалось, что в долларе 100 центов. После такой глобальной замены в долларе оказалось 300 центов, что почему-то не понравилось моему начальнику.

Обратите внимание, что если не включить заголовочный файл cstdlib (или stdlib.h) компилятор посчитает вызов atof() синтаксической ошибкой и напишет, что не знает такой функции. На самом деле это не так. Компилятор знает, что такое библиотечные функции и где они находятся. Что делать, если вы не знаете, какой заголовочный файл нужен для функции atof()? Спросите компилятор. В UNIX можно написать man atof. В Windows — выделить atof в исходном коде и нажать F1 (Help). Появится страница диалогового справочника, рассказывающая о функции atof(), включая то, в каком заголовочном файле она находится.

Определение типов массивов

Во всех предыдущих примерах используемые массивы были массивами-переменными, а не типами. Если нужен другой массив с той же структурой (т. е. типом и числом компонентов), придется определять его заново.

```
double data[NUM];
double tax[NUM];
```

Если эти определения находятся в разных местах программы, то сопровождающему ее программисту (или партнеру разработчика) нужно понимать, что данные объекты имеют одну структуру, т. е. опять встает вопрос написания программного кода так, чтобы знания разработчика можно было передать сопровождающему программисту.

Для этого полезно определить для массива с NUM-компонентами типа double тип SalesData и использовать его затем для определения переменных:

```
SalesData data;
SalesData tax;
```

Такой метод в точности соответствует применению встроенных скалярных типов для определения переменных данного типа. C++ позволяет делать это с помощью ключевого слова typedef. В общем случае typedef определяет новые имена, включая имена типов, исходя из других имен, уже известных компилятору. Оператор, связывающий известный тип и новое имя, ему эквивалентное, имеет вид:

```
typedef известный_тип имя_нового_типа;
```

После этого оператора (завершенного точкой с запятой) программа может использовать известный_тип и имя_нового_типа как синонимы.

Простым примером применения typedef является следующий фрагмент программы, обрабатывающий информацию инвентаризации:

```
int idx, quant, const MAX=30, qty[MAX];
for (idx = 0; idx < MAX; idx++)
  { cin >> quant;
    qty[idx] = quant; }
```

Здесь переменные idx, quant, MAX и элементы массива qty[] — целочисленные, однако это целые другого рода: одна переменная представляет собой индекс массива, другие описывают количество инвентарных единиц. Операторы вида qty[idx] = quant; имеют смысл. Операторы типа idx = quant; смысла не имеют: в вычислениях нельзя смешивать яблоки и апельсины. Что касается правил C++, то для него допустимы оба оператора. Единственный способ подчеркнуть различия — ввести новые имена типов:

```
typedef int Index;                          // один вид целого
Index idx;
const Index MAX=30;
typedef int Stock;
Stock quant, qty[MAX];                      // другой вид целого
for (idx = 0; idx < MAX; idx++)             // сравнение в рамках одного типа
  { cin >> quant;
    qty[idx] = quant; }                     // присваивание в рамках одного типа
```

Здесь idx и MAX имеют один тип — Index, так что их сравнение вполне законно. Переменные quant и qty[idx] также одного типа, поэтому допускается присваивание. Если бы программист написал, к примеру, idx = quant;, то такое присваивание было бы подозрительным, ведь переменные имеют разный тип.

В данном примере для определения нового имени существующего типа int используется typedef. Новый тип не вводится. Они различаются только для программиста. Для компилятора Index и Stock — псевдонимы одного имени типа.

Для определения типа массива используется нечто отличное от директивы typedef, определяющей новый тип:

```
int const MAX = 30;
typedef double SalesData[MAX];
```

В данном определении ключевое слово typedef предшествует синтаксически полному определению массива. При отсутствии ключевого слова typedef это определение привело бы к введению нового имени SalesData. Оно определялось бы через тип double и константу MAX как имя переменной-массива. Поскольку определение typedef присутствует, то вводится имя нового типа SalesData. Согласно тому, что следует за typedef, данный тип определяет массив компонентов MAX типа double. (Конечно, MAX может быть любой константой этапа компиляции, включая литеральное целочисленное значение.)

Теперь можно использовать это имя типа для определения переменных данного типа. Каждая такая переменная представляет собой массив из MAX-компонентов типа double, хотя определение переменной включает в себя только имя типа и имя переменной. Каждое из следующих двух определений массива выделяет память для MAX-компонентов типа double:

```
SalesData data;
SalesData tax;
```

Переменные data и tax могут использоваться как любые переменные-массивы с применением тех же обозначений компонентов массива, как и во всех других случаях:

```
for (int idx = 0; idx < MAX; idx++)
  { tax[idx] = data[idx] * 0.05; }
```

Форма typedef, с помощью которой определялись типы Index и Stock, отличается от формы определения типа SalesData, но они работают одинаково. Определяется новое имя типа — единственный элемент typedef (Index и Stock — в первом случае, SalesData — во втором).

Структуры как неоднородные агрегаты

Существует еще один определяемый программистом тип данных — это структуры. В C++ структуры представляют мощный инструмент для комбинирования связанных компонентов. Они могут определяться с помощью нескольких методов, и мы обсудим наиболее популярные. Все эти методы позволяют программисту определять состав компонентов структуры (поля или компоненты данных), т. е. перечислять их типы и имена.

Структуры, определяемые программистом

Определение структуры начинается с ключевого слова struct, за которым следует назначаемое программистом имя. Оно будет использоваться как имя типа для определения переменных в программе. Поля структуры перечисляются в фигурных скобках, за закрывающей скобкой следует точка с запятой. Объявление каждого поля аналогично объявлению переменной: оно включает в себя тип и определяемое программистом имя, но не включает инициализацию:

```
struct Account {                  // теперь 'Account' - имя типа
  long number;                    // 'number' - имя поля
  double balance;
  double overdue; } ;             // двоеточие за фигурной скобкой
```

Каждое объявление поля начинается с точки с запятой. Если в определении друг за другом следуют поля одного типа, то их можно объявить с помощью одного имени типа и запятых-разделителей. Следующая конструкция определяет в точности такой же тип, что и выше:

```
struct Account {                        // теперь 'Account' - имя типа
  long number;                          // 'number' - имя поля
  double balance, overdue; } ;
```

C++ поддерживает еще и другой синтаксис определения структуры, унаследованный из C. В нем для определения нового типа используется typedef:

```
typedef struct tagAccount {
  long number;
  double balance, overdue; } Account;
```

Такое использование typedef мы уже видели в предыдущем разделе для целых переменных.

```
typedef известный_тип имя_нового_типа;
```

Здесь известный_тип представлен определением struct tagAccount, а Account — это имя_нового_типа (подобно всем предыдущим примерам с typedef, Account — единственное, еще не определенное здесь имя). Фактически struct tagAccount является также именем типа и может использоваться там, где применяется тип Account, однако это имя менее удобно, поскольку ключевое слово struct отличает его от встроенных имен типов. Такая форма определения имен типа для структур очень популярна в C, но в C++ она не нужна.

Создание и инициализация переменных-структур

При определении структуры память не выделяется. Такое определение задает шаблон для будущего распределения памяти: сколько именно памяти нужно выделить, как ее интерпретировать и какие имена использовать для доступа к значениям в памяти. Когда имя структуры используется для определения переменных, это делается точно так же, как в случае встроенных примитивных типов int, double и т. д.

```
Account a1, a2;                    // память для двух переменных Account
```

Здесь создаются две переменные типа Account. Каждая из них содержит поля с именами number, balance и overdue. Общий размер каждой переменной Account равен размеру одного значения long, плюс два значения double, плюс некоторое место для выравнивания (значение не может начинаться с произвольного адреса, а должно выравниваться на адрес, кратный 4 или 8).

Фигурные скобки в определениях типа структуры в предыдущем разделе следует принимать всерьез. Подобно другим областям действия, они обозначают блок со своей отдельной областью действия (об этом подробнее в следующей главе). Определяемые в данной области действия имена вне этой области неизвестны. Так как number — элемент данных типа long, а не переменная long, ее имя нельзя использовать без уточнения:

```
number = 800123456L;               // ошибка
```

Такое выражение неверно, поскольку не указано, к какому счету account принадлежит number. Высокоприоритетная операция точки в C++ (селектор) позволяет выбрать поля переменных-структур. Аналогично компонентам массива, доступным по индексам, к полям структуры можно обращаться как l-значениям и r-значениям с помощью одной и той же формы записи с точкой:

```
a1.number = 800123456L;            // поле используется как l-значение
if (a1.number == 800123456L)       // поле используется как r-значение
  a2.number = a1.number;           // l-значение и r-значение
```

Число элементов структуры должно быть определено во время компиляции. Элементы массива имеют один тип, они не нуждаются в индивидуальных именах, но упорядочены. На элементы массива можно ссылаться с помощью имени переменной-массива и индексов. Компоненты структуры не упорядочены. Они имеют индивидуальные имена и могут быть разных типов. Для ссылки на них используется имя переменной-структуры и имя компонента. Результатом будет значение, тип которого задан для данного компонента в определении структуры.

Поля структуры не упорядочены: определения полей Account могут следовать в любой последовательности. Это никак не повлияет на программу, где данная структура используется.

При создании переменных-структур их поля не содержат полезной информации. Как и в случае с массивами, C++ поддерживает инициализацию структур, где для каждого компонента структуры задается значение соответствующего типа:

```
Account a1 = { 800123456L, 532.84, 0 } ;
```

Такой синтаксис допускается только для структур с общедоступными полями (public), т. е. полями, к которым можно обращаться в клиентском коде (все структуры, описываемые в данной главе, имеют такие поля). Позднее будет рассказано об инициализации структур с закрытыми для доступа полями при помощи конструкторов и переменных-классов.

Как и для неагрегированных переменных, допускается инициализация переменных-структур посредством другой, определенной ранее переменной-структуры:

```
Account a3=a1;              // будет содержать в полях 800123456L, 532.84, 0
```

Иерархические структуры и их компоненты

Цель применения структур C++ состоит в поддержке абстракции данных и инкапсуляции. Поля структуры представляют атрибуты объектов, релевантных приложению: например, данные о сотрудниках, медицинские записи, информация инвентаризации и сведения о заказчиках. Кроме того, они представляют связанные друг с другом данные, которые часто используются совместно: блок управления заданием, таблица символов для синтаксического анализа, структура метрик шрифта, коммуникационный пакет и т. д. Структуры популярны как в системном, так и в прикладном программировании.

Переменная-структура представляет единый составной объект — его компоненты можно использовать индивидуально или как одно целое. С переменными структурного типа разрешается работать целыми блоками и передавать их функциям. Внутри функции компоненты структуры должны обрабатываться отдельно. В свою очередь, переменные-структуры можно комбинировать в массивы, связные списки, очереди и т. д.

```
Account cards[500];                 // массив из 500 структур
```

При обращении к полям, принадлежащим к элементам такого массива, нужно использовать иерархическую запись. Операция индекса и операция точки (селектора) имеют одинаковый приоритет и ассоциируются слева направо. Вот так можно обращаться к полю number компонента с индексом 75 массива cards (код читается справа налево):

```
cards[75].number = 800123456L;
```

Конечно же, структуры можно использовать как компоненты других структур, например скомбинировать фамилию покупателя, адрес и данные счета в новый тип:

```
struct Customer
  {
    char name[30]
    char address[70];
    Account acct;
  } ;
```

Чтобы показать состав структуры, в данном формате фигурные скобки и каждое поле размещаются на отдельной строке. Некоторые программисты предпочитают более сжатый формат:

```
struct Customer
  { char name[30], address[70];
    Account acct; } ;
```

При создании переменной Customer отводимая для нее память содержит два символьных массива и переменную Account, состоящую из поля long и двух полей double. Для элементов массива и элементов Account также используется иерархическая запись:

```
Customer c;
strcpy(c.name, "Doe, John");                    // c.name имеет тип char[]
```

```
strcpy(c.address,"72 Main, Anytown, MA");
c.acct.number = 800123456L;                          // тип long int
c.accnt.balance = 532.841 c.acct.overdue = 0;        // тип double
```

Здесь селектор-точка также ассоциируется слева направо, а читается справа налево, поэтому, например, `c.acct.balance` означает поле `balance` (типа `double`) поля `accnt` (типа `Account`), которые принадлежат переменной-структуре `c` (типа `Customer`).

Отсюда видно, что такая запись может быть довольно громоздкой. Один из способов преодоления данной проблемы состоит в создании функций доступа, упрощающих программный код для работы с агрегированными переменными. Многочисленные примеры функций доступа можно найти ниже.

Операции с переменными-структурами

Переменные-структуры одного типа можно присваивать друг другу. Значения полей исходной переменной по битам копируется в поля целевой переменной:

```
a2 = a1; c.acct = a1;                          // один и тот же тип (Account)
```

Это эквивалентно следующему набору операций присваивания:

```
a2.number = a1.number;
a2.balance = a2.balance; a2.overdue = a1.overdue;
a.acct.number = a1.number; c.acct.balance = a1.balance;
c.acct.overdue = a1.overdue;
```

Здесь вступают в силу свойства C++ как языка с сильным контролем типов. Преобразование между структурами разных типов не допускается. Имена типов должны быть одинаковыми:

```
c = a1; a1 = c;                                // нет, разный тип
a1 = 800123456L;                               // даже и не думайте!
```

Вопрос здесь в имени типа, а не в составе структуры. Предположим, имеется структура с тем же составом, что и `Account`:

```
struct FrozenAcct
{ long number;                                 // та же структура, что и Account
  double balance, overdue; } ;
```

Структуру `FrozenAcct` нельзя присваивать структуре `Account`, и наоборот:

```
FrozenAcct fa; fa = a1;                        // нет, имена типов разные
```

В C++ нет также сравнения структур, так как C++ не знает, какие поля использовать для сравнения и как это делать. Если нужно сравнить переменные структурного типа, придется написать для этого программный код в соответствии с требованиями приложения:

```
if (a1.number > a2.number)                     // поменять счета с номерами заказов
  { a3 = a1; a1 = a2; a2 = a3; }               // a3 содержит временные данные
```

Следующий пример показывает некоторые графические вычисления. Поскольку графические функции не переносимы, в данном примере не используется вывод на экран. Программа просит пользователя ввести координаты конечных точек двух отрезков — AB и CD. Затем вычисляется длина каждого отрезка и угол между отрезком и осью x.

Исходный код этого примера приведен в листинге 5.12. В программе используются два типа — Point и Line. Она запрашивает ввод данных в пикселях (целые значения), инициализирует с их помощью координаты точки, затем инициализирует линии, вычисляет длины и углы. Функции sqrt() и atan2() относятся к заголовочному файлу math.h. Они вычисляют квадратный корень и арктангенс своих параметров типа double. Поскольку их фактические аргументы определены как int, они неявно преобразуются в double. Длина отрезка выражается в пикселях (целые), поэтому результат вычисления квадратного корня неявно преобразуется. Чтобы избежать усечения, для правильного округления к длине добавляется 0,5. Переменная coeff позволяет преобразовать углы из радианов в градусы. На рис. 5.12 показаны результаты выполнения рассматриваемой программы.

```
Введите координаты x и y точки A: 20 20
Введите координаты x и y точки B: 80 80
Введите координаты x и y точки C: 20 20
Введите координаты x и y точки D: 160 80
Длина AB равна 84, а угол равен 45 градусов
Длина CD равна 152, а угол равен 23.1986 градусов
```

Рис. 5.12. *Результат выполнения примера программы из листинга 5.12*

Листинг 5.12. Использование директивы #include для типов, определяемых программистом

```
#include <iostream>
#include <cmath>                                    // для sqrt() и atan2()
#include "point.h"                                  // чтобы тип Point был известен компилятору
#include "line.h"                                   // чтобы компилятору был известен тип Line
using namespace std;

int main ()
{
  const double coeff = 180/3.1415926536;
  Point p1, p2; Line line1, line2;                  // типы, определяемые программистом
  int diffX, diffY, length1, length2;
  double angle1, angle2;
  cout << "Введите координаты x и y точки A: ";
  cin >> p1.x >> p1.y;
  cout << "Введите координаты x и y точки B: ";
  cin >> p2.x >> p2.y;
  line1.start = p1; line1.end = p2;
  cout << "Введите координаты x и y точки C: ";
  cin >> p1.x >> p1.y;
  cout << "Введите координаты x и y точки D: ";
  cin >> p2.x >> p2.y;
  line2.start = p1; line2.end = p2;
  diffX = line1.end.x - line1.start.x;              // неприглядная запись
  diffY = line1.end.y - line1.start.y;
  length1 = sqrt(diffX*diffX + diffY*diffY) + 0.5;
  angle1 = atan(diffY,diffX) * coeff;
  cout  << "Длина AB равна " << length1 << ", а угол равен "
        << angle1 << " градусов\n";
  diffX = line2.end.x - line2.start.x;
  diffY = line2.end.y - line2.start.y;
  length2 = sqrt(diffX*diffX + diffY*diffY) + 0.5;
  angle2 = atan(diffY,diffX) * coeff;
  cout << "Длина CD равна " << length2 << ", а угол равен "
       << angle2 << " градусов\n";
  return 0;
}
```

Определение структур в многофайловых программах

Для небольших программ, как в листинге 5.12, вполне разумно включить определения структур в исходный код. Типы, определяемые программистом, должны быть уникальны в пространстве имен программы. Таким образом, если нужно использовать определяемый программистом тип в разных файлах, программа, состоящая из нескольких файлов, может представлять проблему. Не пытайтесь повторять в файлах определение типа. Если эти определения будут обновляться в процессе сопровождения программы, они легко станут несогласованными.

Решение состоит в том, чтобы поместить каждое определение в отдельный заголовочный файл и включать эти файлы во все исходные файлы программы, где задействован данный тип. Именно это было сделано с типами `Point` и `Line` в листинге 5.12. Обратите внимание на двойные кавычки в именах файлов (вместо угловых скобок). Обычно для заголовочного файла и определяемого в данном файле типа используется одно и то же имя. Поскольку все имена типов в программе C++ обязаны быть уникальными, при хранении файлов в одном каталоге не должен возникать конфликт имен. Часто заголовочные файлы хранятся в отдельном каталоге, отличном от того, где находится выполняемый файл. В этом случае в директиве `#include` следует указать полное имя маршрута.

Обычно, чтобы избежать повторной компиляции определяемого программистом типа данных в многофайловой программе с несколькими исходными файлами, применяют условную компиляцию. Содержимое заголовочных файлов показано в листингах 5.13 и 5.14.

Листинг 5.13. Заголовочный файл `"point.h"`

```
#ifndef _POINT
#define _POINT
struct Point
{ int x, y; } ;
#endif
```

Листинг 5.14. Заголовочный файл `"line.h"`

```
#ifndef _LINE
#define _LINE
#include "point.h"
struct Line
{ Point start, end; } ;
#endif
```

Если включить эти файлы в несколько исходных файлов, то самые первые определения, обработанные компилятором, приведут к определению имен `_POINT` и `_LINE`. При компиляции других файлов будут выполняться директивы условий в этих файлах, исключающие данные определения типов из процесса компиляции. Как правило, для символической константы используется то же имя, что и для типа, только оно записывается в верхнем регистре и с префиксом-подчеркиванием (чтобы избежать случайных конфликтов имен).

Обратите внимание, что файл `"line.h"` должен включать файл `"point.h"`. В противном случае, компилятор может не знать, что означает имя типа `Point` в файле `"line.h"`.

Это все, что требуется знать о структурах для их правильного использования. Основное в применении структур — размещение в каждой структуре только относящейся к ней информации. Старайтесь избегать включения в структуру полей,

не соотносящихся с другими полями. Еще один важный момент: каждая переменная-структура должна иметь полное определение полей, доступных с помощью записи с точкой и объявленных имен. Наконец, каждое поле (в записи с точкой) представляет собой просто переменную типа, заданного для этого поля в определении структуры.

Объединения, перечисления и битовые поля

Данный раздел должен быть относительно короток. В нем обсуждаются три идеи, связанные с именованием программных элементов для удобства программиста. Первая идея состоит в определении переменной, которую можно использовать для хранения информации нескольких типов, например целого и числа с плавающей точкой. Именно такова идея *объединения* — union. (В отличие от структуры, его объекты могут иметь различные типы в разные моменты времени.— *Прим. пер.*) Вторая идея заключается в определении символических имен для родственных констант, не вдаваясь в детали присваивания этим символическим константам числовых значений. Такова идея *перечисления* — enum. Третья идея состоит в именовании фрагментов данных, чтобы с ними можно было работать отдельно. Это *битовые поля*.

В C++ данные идеи реализуются аналогично структурам. Программист использует в начале определения типа ключевое слово (`union`, `enum` или `struct`). Затем он вводит выбранное для нового типа имя и описывает состав типа (в фигурных скобках с завершающей точкой с запятой). После этого определенное программистом имя может использоваться в программе как имя типа.

Объединения

Рассмотрим массив структур типа `Number` с некоторой произвольной числовой информацией.

```
Number num[6];
```

Поскольку любое число является здесь допустимым, следует использовать нечисловое контрольное значение, например строку "end" или что-то еще. Поддерживаемый языком C++ строгий контроль типов не позволяет хранить текстовую информацию в числовом поле. Возможный вариант решения данной проблемы — в определении структуры с двумя полями, одно из которых содержит числовое значение, а другое текст. Определение имеет следующий вид:

```
struct Number
{ double value;
  char text[4]; } ;
```

Теперь можно использовать элементы этого типа в массиве, сохраняя в поле `value` числовую информацию и используя поле `text` для контрольного значения (например "end"), указывающего конец допустимых данных в массиве. Между тем для каждого элемента массива используется только одно поле. Элемент является либо допустимым числовым значением, либо текстовым контрольным. С помощью определения `union` C++ позволяет избежать нерационального расходования памяти. Ключевое слово `union` определяет новый тип. При этом используется тот же синтаксис, что и при определении структур. Если полей несколько, они дают альтернативное представление одной и той же области в памяти. Вот как будет выглядеть определение `union` в данном примере:

```
union Number              // еще одно ключевое слово C++
{ double value;           // можно определять любое число полей
  char text[4]; } ;       // не забывайте про точку с запятой!
```

В этом определении вводится тип Number. Можно определять переменные такого типа — каждая из них будет состоять из двух полей. В отличие от структур, эти поля не существуют одновременно, а представляют разные интерпретации одной и той же области в памяти. Когда определяется переменная union, выделяется столько памяти, чтобы вместить ее самую длинную интерпретацию. Затем программа может выбирать, какую именно интерпретацию использовать: с плавающей точкой или массив символов. Если кто-то сделает ошибку и сохранит данные одного типа, а затем прочитает их как другой тип, то результат будет бессмысленным. Как обычно, компилятор C++ не проверяет за программиста, правильно ли он работает с памятью.

В следующем примере числовое значение помещается в переменную-объединение n1, а строка — в переменную-объединение n2, но затем содержимое n1 отображается на экране как текст, а содержимое n2 — как числовое значение. Нехорошо, конечно, но ведь это свободная страна. Пример показывает также, что можно вполне законно хранить текст в том месте, где используется числовое значение, и наоборот:

```
Number n1, n2;
n1.value = 5.0; strcpy(n2.text,"no");          // принятие обязательств
cout << n1.value << " " << n2.text << endl;    // это нормально
cout << n1.text << " " << n2.value << endl;    // катастрофа
strcpy(n1.text, "yes"); n2.value = 25.0;       // старое содержимое исчезает
cout << n1.text << " " << n2.value << endl;    // теперь OK
```

```
11
21
31
end
Text as double: 3.57452e-031
```

Рис. 5.13. *Вывод программы из листинга 5.15*

Пример показывает, что при работе с переменными-объединениями используется запись, аналогичная операциям со структурами. В листинге 5.15 представлена иллюстрирующая это короткая программа. Все выглядит так, будто поля text первых трех компонентов массива num[ ] не были инициализированы, как и поле value последнего использованного компонента. На самом деле обе интерпретации требуют одинакового объема памяти. (Компилятор выделяет для каждого элемента одну и ту же память — достаточную для последней интерпретации.) Вывод программы показан на рис. 5.13.

Листинг 5.15. Применение union для хранения в переменной значений разных типов

```
#include <iostream>
#include "number.h"                          // чтобы тип Number был известен компилятору
#include <cstring>
using namespace std;

int main ()
{
  Number num[6]; int i = 0;                  // массив переменных union
  num[0].value = 11.0; num[1].value = 21.0;  // инициализация
  num[2].value = 31.0; strcpy(num[3].text,"end");
  while(strcmp(num[i].text, "end") != 0)     // итерация
    cout << num[i++].value << endl;
  cout << num[i].text << endl;               // для иллюстрации
  cout << "Текст как double: " << num[i].value << endl;

  return 0;
}
```

Пусть вас не пугает обозначение num[0].value. Массив num[] содержит компоненты типа Number, следовательно, num[0] имеет тип Number и содержит поля с именами value и text. Когда используется поле text, это символьный массив, т. е. можно передать num[i].text в качестве аргумента функции strcmp(). Увеличение индекса при выводе содержимого поля (как в num[i++].value) вполне законно и целесообразно. Так как индекс i используется здесь только один раз, порядок вычисления нарушен не будет.

Две последние строки программы показывают, как выглядит одно и то же значение при разной интерпретации. Ошибочное использование объединений ведет к получению бессмысленных данных. Заметим, что значение 3.57452e–031 может быть в программе вполне законным значением с плавающей точкой и интерпретироваться как "end" для завершения итерации.

Чтобы избежать ошибок в интерпретации, некоторые программисты применяют в объединениях так называемые поля тегов. Теговое поле не может быть компонентом объединения, поэтому объединение и теговое поле нужно включить в более крупную структуру. Например, пусть адрес содержит три строки, вторая и третья строка — это дом/название улицы и номер почтового отделения. В листинге 5.16 приведена программа, определяющая адрес структуры с полем объединения second и теговым полем kind. Когда теговое поле равно 0, вторая строка интерпретируется как дом и название улицы, а когда она равна 1 — как номер почтового отделения. В программе соблюдается данное соглашение при установке значений данных (присваиванием kind значения 0 и 1) и при использовании данных (проверкой значения поля kind). Вывод программы представлен на рис. 5.14.

```
Doe, John
15 Oak Street
Anytown, MA 02445

King, Amy
P.O.B. 761
Anytown, MA 02445
```

Рис. 5.14. *Вывод программы из листинга 5.16*

Листинг 5.16. Использование объединения с полем тега для обеспечения целостности данных

```
#include <iostream>
#include <cstring>
using namespace std;

union StreetOrPOB
   { char street[30];                          // альтернативное представление
     long int POB; } ;

struct Address
{  char first[30];
   int kind;                                   // 0 - дом/улица, 1 - почтовое отделение
   StreetOrPOB second;                         // тот же или другой смысл
   char third[30]; } ;

int main ()
{
   Address a1, a2;
   strcpy(a1.first,"Doe, John");               // дом/улица
   strcpy(a1.second.street,"15 Oak Street"); a1.kind = 0;
   strcpy(a1.third,"Anytown, MA 02445");
   strcpy(a2.first,"King, Amy");
   a2.second.POB = 761; a2.kind = 1;           // адрес с почтовым отделением
strcpy(a2.third,"Anytown, MA 02445");
   cout << a1.first << endl;
   if (a1.kind == 0)                           // проверка интерпретации данных
      cout << a1.second.street << endl;
```

```
    else
      cout << "P.O.B " << a1.second.POB << endl;
    cout << a1.third << endl;
    cout << endl;
    cout << a2.first << endl;
    if (a2.kind == 0)                          // проверка интерпретации данных
      cout << a2.second.street << endl;
    else
      cout << "P.O.B. " << a2.second.POB << endl;
    cout << a2.third << endl;
    return 0;
}
```

Все хорошо, но приходится вводить еще один уровень иерархической структуры типов. В результате программисту придется применять такие имена, как a1.second.street, и это не радует. Между тем, единственное назначение типа StreetOrPOB — использование с типом Address. Избавиться от этого можно с помощью анонимных объединений C++. У них нет имени, и переменные данного типа определять нельзя, однако к полям таких объединений можно обращаться без каких-либо уточнений. Например, удобно определить тип Address не с помощью типа StreetOrPOB, а посредством анонимного объединения:

```
struct Address
{ char first[30];
  int kind;                    // 0 - адрес (улица), 1 - почтовый ящик
  union
  { char street[30];
    long int POB; } ;          // нет "второго" (second) поля типа StreetOrPOB
  char third[30]; } ;
```

Тип объединения исчез, но тип Address имеет теперь два альтернативных поля — street[] и POB. Как и на любое другое поле, на них можно ссылаться по имени. Конечно, программист должен знать, что есть что. Мухи отдельно, котлеты отдельно. Данные должны считываться корректно, установленным способом. Однако дополнительный уровень иерархии теперь не нужен:

```
if (a1.kind == 0)                          // проверка интерпретации данных
  cout << a1.street << endl;               // использование одной интерпретации
else
  cout << "P.O.B. " << a1.POB << endl;     // или применение другой
```

Это мощный метод программирования, однако сопровождающему приложение программисту потребуется дополнительное время и усилия, чтобы разобраться в исходном коде. Увеличение числа операторов условия усложняет программу. Возможно, применение наследования с виртуальными функциями станет хорошей альтернативой данному методу, но об этом позднее.

Перечисления

Тип "перечисление" позволяет программисту определять переменные, в которых сохраняются значения только из определенного набора идентификаторов. Обычно вводятся целочисленные символические константы (с помощью #define или определений const) и принимается соглашение по их использованию. Например, для эмуляции поведения светофора нужны значения, обозначающие красный, зеленый и желтый свет. Как в примере с днями недели, можно ввести массив

символов "red" (красный), "green" (зеленый) и "yellow" (желтый), а затем для присваивания и сравнения использовать библиотечные функции.

```
char light[7] = { "green" };                // сначала - зеленый
if (strcmp(light, "green") == 0             // далее - желтый
  strcpy(light, "yellow");                  // и т. д.
```

Хорошо и понятно — при сопровождении программы не будет проблем с интерпретацией намерений разработчика, но эти операции со строками выполняются довольно медленно. Зачем перемещать столько символов (в библиотечной функции в поиске конца строки) только для того, чтобы отследить состояние светофора? Еще одним недостатком такого решения является отсутствие защиты. Если кто-то захочет сделать свет розовым или малиновым, то у программиста нет способа помешать этому.

Другое решение состоит в использовании целых чисел, обозначающих цвета по номерам. Можно присвоить 0 — зеленому, 1 — красному, 2 — желтому. Обратите внимание, как вводятся эти значения: 0, 1 и 2, а не 1, 2 и 3. Это связано с особенностями массивов C++ и индексов. Когда программист, привыкший писать программы на C++ или на C, считает людей в комнате, он говорит: "Ноль, один, два...".

Такой подход позволяет избежать использования функций работы со строками:

```
int light = 0;                  // сначала - зеленый
if (light == 0)                 // следующий - желтый
  light = 2;                    // и т. д.
```

Преимуществом такого подхода является скорость, но это единственное его преимущество. Такой стиль программирования всегда требует комментариев, особенно в сложных алгоритмах и при большом числе состояний. Если комментарии слишком загадочные или устаревшие, то сопровождающему приложение программисту придется, мягко говоря, не просто.

Один из способов сделать код более читабельным и при этом сохранить высокую скорость работы программы состоит в использовании символических констант. Можно определить символические константы с подходящими для приложения именами, например RED, GREEN и YELLOW, а затем присвоить им целые значения.

```
const int RED=0, GREEN=1, YELLOW=2;          // константы цветов
```

Теперь нетрудно переписать приведенный выше пример с помощью данных констант. Такой код выполняется так же быстро, как и предыдущий, но при этом столь же понятен, как пример с символьными строками:

```
int light = GREEN;                  // сначала - зеленый
if (light == GREEN)                 // следующий - желтый
  light = YELLOW;                   // и т. д.
```

Подобное решение защитит программу от внесения ошибок при сопровождении. Если программист захочет вместо символических констант использовать числа, это будет синтаксическая ошибка. Однако если переменной light будет присвоено значение вне согласованного для набора цветов диапазона (например, light = 42), то синтаксической ошибки не будет. Можно складывать эти значения (RED + GREEN) и выполнять разные другие операции, которые невозможно выполнять с цветами.

Перечисления введены в язык как раз для решения таких проблем. Программист может определить тип и явно перечислить все допустимые для переменной значения. Для такого определяемого программистом типа (например, Color) используется ключевое слово enum и делается это аналогично применению ключевого слова struct (или union). Перед фигурными скобками (за которыми ставится

точка с запятой) указывается имя типа (подобно struct и union). В фигурных скобках программист перечисляет все допустимые для определяемого типа значения. Часто программисты используют буквы в верхнем регистре (как для констант, определяемых в #define или в const). Например, нетрудно определить тип Color как enum:

```
enum Color { RED, GREEN, YELLOW } ;       // цвет как тип
```

Теперь можно использовать тип Color для определения переменных, для которых допускаются только значения RED, GREEN, YELLOW. Эти значения являются перечислимыми константами. Их можно применять только как неизменяемые r-значения:

```
Color light = GREEN;                      // сначала - зеленый
if (light == GREEN)                       // далее - желтый
  light = YELLOW;                         // и т. д.
```

Такое решение устраняет проблему: для значений перечисления допустимы только операции присваивания и отношения. Их нельзя использовать для ввода-вывода, но можно проверять на равенство или неравенство, сравнивать (больше, меньше):

```
if (light > RED) cout << "True\n";        // выводит 'True'
```

Причина в том, что перечисления реализованы как целые. Первое значение в списке перечисления — 0 (что для C++ неудивительно), далее 1 и т. д. Данная программа может обращаться к этим значениям путем приведения их типа (как значений перечислимого типа) к целым:

```
cout << (int) light << endl;              // выводит 0, 1 или 2
```

Если программист захочет изменить данное значение на другое, то можно сделать это явно в списке перечисления:

```
enum Color { RED, GREEN=8, Yellow } ;         // YELLOW теперь равно 9
```

После этого присваивание значений возобновляется (YELLOW равно 9 и т. д.). Если по какой-то причине программист захочет установить GREEN в 0, то программа не сможет различить RED и GREEN (не проблема, если не надо управлять движением на дорогах).

Такая техника полезна, когда значения перечисления планируется использовать как маски для поразрядных операций и, следовательно, они должны представляться степенью двойки:

```
enum Status { CLEAR = 2, FULL = 8, EMPTY = 64 } ;
```

Многие программисты с энтузиазмом воспримут такое средство и возможность его использования для определения констант этапа компиляции:

```
enum {SIZE = 80 } ;              // используйте для определения массивов и пр.
```

Обратите внимание, что это перечисление анонимно (аналогично анонимному объединению). Оно не имеет имени и, следовательно, нельзя определять переменные данного типа. Однако невелика потеря, поскольку все, что здесь нужно — это символическая константа SIZE. Результат будет тот же, что и при явном определении константы:

```
const int SIZE = 80;          // то же самое
```

Битовые поля

Как и при обсуждении объединений и перечислений, разговор о битовых полях лучше начинать с примеров практических проблем, которые можно решить с помощью дополнительных типов C++.

Наименьшим объектом, доступным для распределения и адресации в программе C++, является символ. Иногда в программе может потребоваться значение меньшего размера, в этом случае резервирование для него полноразмерного целого было бы напрасной тратой памяти. Кроме того, часто форматы внешних данных и интерфейсы аппаратных устройств вынуждают обрабатывать отдельные элементы слова.

Например, контроллер дискового массива может манипулировать с адресами памяти и их компонентами: номером страницы (от 0 до 15) и смещением адреса памяти на странице (от 0 до 4095). Данный алгоритм может потребовать операций с номерами страниц (4 бита), смещением (12 бит) и полным адресом (16 бит без знака), комбинирования номера страницы и смещения в адрес, извлечения из адреса номера страницы и смещения.

Еще одним примером может быть порт ввода-вывода, где конкретные биты ассоциируются с заданными условиями и операциями. Бит 1 порта может устанавливаться, если устройство свободно для передачи состояния, бит 3 — если заполнен буфер приема, а бит 6 — если пуст буфер передачи. Алгоритм может потребовать индивидуальной установки каждого бита в слове состояния и считывания состояния каждого бита. Для каждой из этих вычислительных задач нужно использовать поразрядные логические операции.

Комбинирование номера страницы и смещения в адрес памяти требует сдвига адреса на 12 позиций влево и выполнения с результатом сдвига и адресом поразрядной операции ИЛИ.

```
unsigned int address, temp;    // должны быть беззнаковыми
int page, offset;              // бит знака никогда не устанавливается в 1
temp = page << 12;             // сделать четыре бита старшими
address = temp | offset;       // предположим, дополнительных битов нет
```

Получение из адреса номера страницы и смещения — более сложная задача. Чтобы получить номер страницы, нужно сдвинуть адрес на 12 позиций, отбросив тем самым биты смещения, и переместить номер страницы в младшие биты слова. Для получения адреса используется поразрядная операция AND и маска 0x0FFF, которая устанавливает каждый из 12 младших бит в 1:

```
page = address >> 12;          // отделить биты смещения, получить биты страницы
offset = address & 0x0FFF;         // отделить биты страницы от адреса
```

Чтобы установить отдельные биты в 1, используются три маски: у каждой маски только 1 бит установлен в 1, а все другие равны 0. Применяя к слову состояния поразрядную операцию ИЛИ, можно установить соответствующий бит в 1, если он равен 0, или оставить биты в прежнем состоянии, если они уже равны 1. Определенные выше константы CLEAR, FULL и EMPTY представляют собой маски, у которых только 1 бит установлен в 1, а другие биты нулевые. У константы CLEAR бит 1 установлен в 1, у FULL бит 3 установлен в 1, у EMPTY шестой бит — в 1.

```
unsigned status=0;             // предположим, инициализируется правильно
status |= CLEAR;               // установить бит 1 в 1 (если он нулевой)
status |= FULL;                // установить бит 3 в 1 (если он нулевой)
status |= EMPTY;               // установить бит 6 в 1 (если он нулевой)
```

Для сбрасывания отдельных бит в 0, потребуются маски, где все биты, за исключением одного, установлены в 1. При использовании поразрядной логической операции "И" все биты, кроме нулевого бита в маске, остаются неизмененными. Чтобы сбросить бит 1, нужна маска с нулевым первым битом. Для сброса третьего бита потребуется маска, у которой третий бит установлен в 0. Чтобы сбросить бит 6, нужно применить маску с нулевым шестым битом и остальными битами, установленными в 1. Эти маски трудно выразить в виде десятичных или шестнадцатеричных констант. Кроме того, на разных платформах могут потребоваться маски разного размера, что влияет на переносимость кода. Очень часто для установки битов в 1 используют отрицание (инверсию). С помощью операции "И" можно сбросить эти биты в 0:

```
status &= ~CLEAR;                    // сброс бита 1 в 0 (если он равен 1)
status &= ~FULL;                     // сброс бита 3 в 0 (если он равен 1)
status &= ~EMPTY;                    // сброс бита 6 в 0 (если он равен 1)
```

Для доступа к значениям отдельных битов используется операция "И" с масками, где все биты установлены в 0, за исключением одного, к которому нужно обратиться. Если данный бит установлен, то результат операции будет ненулевым (`true`), а если он установлен в 0, то результатом будет 0 (`false`). С этими операциями будут работать такие же маски, какие применялись для установки и сброса бита состояния:

```
int clear, full, empty;              // для проверки на true или false
clear = status & CLEAR;              // true, если бит 1 установлен в 1
full = status & FULL;                // true, если бит 3 установлен в 1
empty = status & EMPTY;              // true, если бит 6 установлен в 1
```

Эти операции нижнего уровня для упаковки и распаковки последовательностей битов (пример с адресацией) или отдельных битов (пример с состоянием) достаточно сложны, часто приводят к ошибкам, и их нельзя назвать интуитивно понятными. C++ позволяет присваивать имена сегментам битов разного размера. Это делается с помощью обычного определения структуры. Для каждого поля задается число битов (длина поля) — после двоеточия указывается неотрицательная константа:

```
struct Address {
  int page : 4;
  int offset : 12; } ;                // не очень велико для 12 бит
```

Поля данных упакованы в целое. С целыми со знаком нужно быть поаккуратнее — один бит обычно отводится для знака. Если нужно использовать все выделенные для поля биты, то поле должно быть беззнаковым:

```
struct Address {
  unsigned int page : 4;
  unsigned int offset : 12; } ;       // место для 12 бит
```

Поле может не вписываться в границы слова. Если оно не помещается в машинном слове, то выделяется следующее слово, а оставшиеся биты не используются. Если размер поля превышает размер базового типа на данной платформе (который может быть разным на разных машинах), то будет синтаксическая ошибка.

Поля позволяют сэкономить память: нет необходимости выделять для каждого значения байт или слово, однако размер оперирующего с данными значениями программного кода увеличивается, поскольку требуется извлекать биты. Конечный результат неясен.

Эти переменные определяются точно так же, как переменные-структуры. Доступ к битовым полям соответствует доступу к обычным полям структуры:

```
Address a; unsigned address;                    // должно инициализироваться
address = (a.page << 12) | a.offset;
```

Если нужно выделить для флага 1 бит, убедитесь, что это поле unsigned, а не signed. Поля не обязательно должны иметь имена. Неименованные поля используются для дополнения (но при этом все равно нужно задавать тип, указывать двоеточие и длину поля):

```
struct Status {
  unsigned : 1;                 // бит 0
  unsigned Clear : 1;           // бит 1
  unsigned : 1;                 // бит 2
  unsigned Full : 1;            // бит 3
  unsigned : 2;                 // бит 4 и 5
  unsigned Empty : 1;};         // бит 6
```

Написать код для операций с переменными состояния очень просто. В своей основе он реализуется через сдвиги и поразрядные логические операции, аналогично примерам, уже приведенным в начале раздела.

```
Status stat;                                  // убедитесь в инициализации
int clear, full, empty;                       // для тестирования на true или false
stat.Clear = stat.Full = stat.Empty = 1; // установить бит в 1
stat.Clear = stat.Full = stat.Empty = 0; // сбросить биты в 0
clear = stat.Clear;                           // значения могут быть протестированы
full = stat.Full;
empty = stat.Empty;
```

Допускается нулевая длина поля. Это указывает компилятору, что следующее поле нужно выровнять на границу целого. Разрешается смешивать данные различных целых типов. Переключение типа с одного размера на другой выделяет следующее поле в памяти. Как показывает приведенный ниже пример, неаккуратное использование битовых полей может привести к тому, что выделяемое пространство не уменьшится (данный код написан на 16-разрядной машине, где для целых выделяется два байта):

```
struct Waste {
  long first : 2 ;                  // здесь выделяются все 4 байта
  unsigned second : 2 ;             // здесь добавляются еще два
  char third : 1;                   // short начинается на четном адресе
  short fourth : 1; } ;             // всего 10 байт
```

На некоторых машинах поля присваиваются слева направо, а на других — справа налево (это называется обратный и прямой порядок байтов).

Для внутренних структур данных порядок полей не проблема, однако для отображения внутреннего определения данных, например буферов устройства ввода-вывода, имеет значение. Когда "внешние" данные поступают в одном формате, а компилятор использует другой, данные в битовых полях могут сохраняться некорректно.

Прежде чем использовать битовые поля, следует рассмотреть альтернативные варианты. Не забывайте, что доступ к целому или символу всегда происходит быстрее, чем доступ к битовому полю, а требует меньше исходного кода.

Итоги

В данной главе рассматривались основные средства построения программ, которые программист может использовать для создания крупных приложений. Многие из них относятся к агрегированию данных в более крупные блоки: однородные (массивы) и неоднородные объекты (структуры). За исключением присваивания структур, операций для таких агрегатных типов данных не предусматривается. Все операции с подобными объектами должны программироваться на уровне операций с отдельными элементами.

Так как к полям структур можно обращаться с помощью отдельных имен полей, это относительно безопасно. Компоненты массивов доступны с помощью индексов, и программа C++ не предусматривает защиты на этапе компиляции или выполнения от недопустимых значений индексов, что может легко привести к некорректным результатам или порче памяти — частый источник проблем для программистов. Это особенно относится к символьным массивам, где конец допустимых данных задается завершающим нулем (нулевым терминатором).

Здесь рассматривались такие определяемые программистом типы данных как объединения, перечисления и битовые поля. В отличие от массивов и структур, они не столь необходимы. Любую программу можно написать без применения этих элементов, однако они часто упрощают вид программы, позволяют лучше передать идеи разработчика и облегчают работу по сопровождению ПО.

Глава 6

Управление памятью: стек и динамически распределяемая область

Темы данной главы

- ✔ Область действия имени как средство кооперации
- ✔ Управление памятью: классы памяти
- ✔ Управление памятью: использование динамически распределяемой области
- ✔ Обмен данными с файлами на диске
- ✔ Итоги

В предыдущей главе рассказывалось о средствах реализации структур данных, определяемых программистом. Массивы и структуры являются базовыми средствами программирования, позволяющими разработчикам выражать в приложении сложные идеи, причем в понятной (как самим разработчикам, так и специалистам по сопровождению ПО) форме. Объединения, перечисления и битовые поля помогают разработчикам представить исходный код программы в наиболее легко интерпретируемом виде.

Все переменные встроенного или определяемого программистом типа, использовавшиеся в предыдущих главах, были именованными. Программист должен выбирать имя переменной и включать в исходный код ее определение. Когда программе требуется память для именованных переменных, она выделяет и освобождает ее согласно правилам языка, без дальнейшего участия программиста. Память выделяется в области, называемой *стеком*. За эту простоту приходится расплачиваться отсутствием гибкости. Размер каждого элемента данных определяется на этапе компиляции.

Для получения гибких структур данных C++ позволяет программисту создавать динамические массивы и связанные структуры данных, однако это вынуждает использовать указатели и увеличивает сложность программы. Когда требуется дополнительная память для динамических неименованных переменных, она берется из так называемой *динамически распределяемой области* (heap). Динамические переменные не имеют имен, и ссылаться на них нужно косвенно, через указатели. Тем самым за счет достаточно сложного динамического управления памятью обеспечивается более высокая гибкость.

В данной главе рассказывается о методах C++ для управления стеком и динамически распределяемой областью, о базовых приемах использования пространства имен, экстента имен и динамического управления памятью с помощью указателей. Эти методы имеют ключевое значение для эффективного использования системных ресурсов, однако в неопытных руках динамическое управление памятью может легко привести к аварийному завершению работы системы, порче содержимого памяти, "утечкам" памяти (когда системе в итоге не хватает памяти, потому что она правильно не освобождается). Некоторым программистам по вкусу мощные возможности и увлекательность динамического управления памятью, другие предпочитают применять указатели как можно меньше. Каковы бы ни были ваши личные предпочтения, нужно хорошо понимать принципы управления именами и памятью, поддерживаемые в C++.

Перед обсуждением вопросов динамического управления памятью нужно познакомиться с принципами области действия имен и классами памяти. Это важно для понимания особенностей управления памятью в C++. После рассмотрения вопросов динамического управления памятью мы перейдем к методам использования внешней памяти — файлам на диске. Хранение данных в файлах позволяет программе работать практически с неограниченными по объему наборами данных.



Внимание Эта большая глава содержит целый набор важных концепций и практических методов программирования. Нельзя стать квалифицированным разработчиком программ C++ без освоения основных концепций и принципов управления файлами и ввода-вывода для обмена данными с файлами. Но можно изучать остальную часть C++, даже не став экспертом в данных областях. Если вас пугает размер и сложность этого материала, перейдите к следующей главе и вернитесь к главе 6, когда почувствуете, что готовы на большее.

Область действия имени как средство кооперации

Каждое определяемое программистом имя (идентификатор) имеет в программе на языке C++ лексическую область действия (или просто область действия).

"Лексическая" она потому, что ссылается на сегмент исходного кода, где имя известно и может использоваться, а "область действия", поскольку вне данного сегмента программного кода имя либо неизвестно, либо ссылается на что-то совершенно другое. Элементы программы, имена которых имеют область действия,— это имена определяемых программистом типов данных, функций, параметров, переменных и меток. Такие имена можно использовать в выражениях, определениях, вызовах функций и т. д.

Лексические области действия в C++

Лексическая область действия — статическая характеристика имени. Она означает, что область действия определяется лексической структурой программы на этапе компиляции, а не поведением программы во время выполнения. В C++ есть следующие области действия:

- Блока
- Функции
- Файла
- Всей программы
- Класса
- Пространства имен

В этой главе обсуждаются первые четыре области. О других двух будет рассказано после более детального ознакомления с принципами классов и пространств имен. Область действия блока ограничивают открывающая и закрывающая фигурные скобки. Область действия функции также ограничивается открывающей и закрывающей фигурной скобкой. Разница между ними в том, что функция имеет параметры (и их имена известны в ее области действия). Вход в область действия функции осуществляется при выполнении программы, когда эта функция вызывается. Блок не вызывается, а выполняется после предшествующих ему операторов (если такие есть). Например, при итерации в цикле `for` область действия неименованного блока ограничивается фигурными скобками и при каждой итерации программа входит в эту область. При вызове функции `getBalance()` вызывается (с помощью имени функции) область действия ее блока (эта функция приведена ниже в листинге 6.1):

```
for (i = 0; i < count; i++)
{ total += getBalance(a[i]); }              // накопление итога
```

Область действия файла ограничивается физическими границами файла. Она содержит определения типа, определения и описания переменных, определения и описания функций. Каждый листинг программы, использовавшийся в предыдущих главах, представлял собой листинг исходного файла в границах этого файла.

Область действия программы ограничителей не имеет. Все, что принадлежит исходному файлу, входящему в состав программы, является частью ее области действия.

Конфликты имен в одной области действия

Конфликты имен внутри области действия в C++ не допускаются. Имя должно быть уникальным в той области действия, где оно описано. В языке C определяемые программистом типы обычно формировали отдельную область действия. Это означает, что если имя использовалось для типа, оно могло применяться в той же области действия для переменной. Компилятор (и сопровождающий приложение программист) из контекста делали вывод, что означает имя — тип или переменную.

В C++ реализуется более строгий подход. Все определяемые программистом имена образуют отдельное пространство имен. Если имя определяется в области действия для каких-либо целей, оно должно быть в ней уникально, т. е. отличаться от всех объявленных (для каких-то целей) в той же области действия имен. Это означает, что, если, к примеру, `count` — имя переменной, то ни тип, ни функция, ни параметр, ни другая переменная не может называться так в той области действия, где объявлена переменная `count`.

Подобно большинству других подходов программной инженерии, связанных с языками программирования, эта идея нацелена на улучшение читабельности и простоты написания программ. Когда разработчик или программист находит в исходном файле имя `count`, то не нужно гадать, какой смысл оно несет: в данной области действия у него только один смысл. Если разработчик хочет включить в область действия переменную `count`, он должен посмотреть, нет ли в ней другого такого же имени.

Единственное исключение из этого правила — имена меток. Они не конфликтуют с именами переменных, параметрами или типами, описанными или известными в той же области действия. Так как метки применяются в C++ нечасто, от них программу не становится сложнее читать, но злоупотреблять ими не следует.

Из этого принципа вытекает и обратное: одно и то же имя может использоваться в разных областях действия без конфликтов, что уменьшает необходимость координации между разработчиками. Разные программисты могут работать над разными частями программы (разными файлами) и выбирать имена независимо. Взаимодействие между членами команды программистов для этого не требуется.

В одном файле нужно координировать имена, определенные в разных областях действия, что осложняет задачу разработчика.

Лексические области действия разных элементов программ (типов данных, функций, параметров, переменных и меток) в чем-то различны. Имена блоков могут объявляться в блоке, функции или файле. Они известны внутри этого блока, функции или файла с места определения до конца области действия. Вне области действия данного блока, функции или файла они неизвестны. То же относится к именам переменных. Они могут объявляться в блоке, функции или файле и известны с точки определения до конца области действия.

Параметры могут определяться только в функции. Они известны с открывающей фигурной скобки функции до ее закрывающей фигурной скобки. Метки определяются в блоке или в функции, но их имена известны во всей функции, использующей метку, и неизвестны вне функции.

Имена функций C++ могут определяться в файле, но не в блоке или в другой функции. Областью действия имен функций является программа, то есть имя функции должно быть уникально в проекте. Необходимость координации имен в масштабе проекта часто усложняет координацию в коллективе разработчиков. То же относится и к расширению существующих программ при их сопровождении: при добавлении новых имен функций возможны конфликты. Еще один потенциальный источник неприятностей, связанный с именами функций, это интеграция нескольких библиотек от разных поставщиков (или из прежних проектов). Часто подобная проблема может не проявляться, пока созданные разными программистами файлы не будут скомпонованы на последних этапах разработки.

```
Данные загружены

800123456
800123123
800123333
1200
1500
1800

Остаток на счете в долл. 4500
```

Рис. 6.1. *Вывод программы из листинга 6.1*

В листинге 6.1 приведен простой пример загрузки учетных данных, их отображения и вычисления балансовых сумм. Для простоты данные здесь не вводятся с клавиатуры и не загружаются из внешнего файла или БД. (Этим мы займемся позднее.) Используется просто два массива `num[]` и `amounts[]`, которые подставляют значения учетных и балансовых данных. Сами данные загружаются в бесконечном цикле `while`, пока не обнаруживается контрольное значение –1. Затем второй цикл выводит номера счетов, третий — показывает балансовые данные, а четвертый — балансовые суммы. Здесь применяются два определенных программистом типа — структура `Account`, целочисленный синоним `Index` и функция `getBalance()`. Они используются не столько в силу необходимости, сколько для иллюстрации области действия. Для простоты набор данных берется небольшой. Вывод программы представлен на рис. 6.1.

Листинг 6.1. Демонстрация лексической области действия типов, параметров и переменных

```
#include <iostream>
using namespace std;

struct Account {                                    // глобальное определение типа
   long num;
   double bal; } ;

double getBalance(Account a)
{ double total = a.bal;                             // total в независимой области действия
   return total; }                                  // возвращает a.bal (лучше)

int main()
{
   typedef int Index;                               // локальное определение типа
   Index const MAX = 5;
   Index i, count = 0
   Account a[MAX]; double total = 0;                // набор данных и итоговое значение
```

```
  while (true)                                        // выход по контрольному значению
  { long num[MAX] = { 800123456, 800123123, 800123333, -1 } ;
    double amounts[MAX] = { 1200, 1500, 1800 } ;      // данные для загрузки
    if (num[count] == -1) break;                      // найдено контрольное значение
    a[count].num = num[count];                        // загрузка данных
    a[count].bal = amounts[count];
    count++; }
    cout << " Данные загружены\n\n";
  for (i = 0; i < count; i++)
    { long temp = a[i].num;                           // temp в независимой области действия
      cout << tmp << endl; }                          // вывод учетных номеров
  for (i = 0; i < count; i++)
    { double temp = a[i].bal;                         // temp в независимой области действия
      cout << tmp << endl; }
  for (i = 0; i < count; i++)
    { total += getBalance(a[i]); }                    // накопление total для балансов
  cout << endl < " Остаток на счете в долл." << total << endl;
  return 0;
}
```



Внимание Данная программа компилировалась с помощью последней версии 32-разрядного компилятора, поэтому нет необходимости указывать, что значения 800123456 и другие имеют тип long. 16-разрядным компилятором эта программа компилироваться не будет. Аналогично примерам главы 5, здесь для значений использовался суффикс L (800123456L и т. д.). Такие примеры подходят для любого компилятора. Программисты, применяющие C++, всегда должны продумывать вопросы переносимости, в противном случае возможны ошибки, поиск и исправление которых стоит времени и денег.

Здесь тип Account действует во всем файле и известен с места его определения до конца исходного файла. Переменные типа Account могут определяться в любом месте этой области действия. Применение имени Account в данной области действия для любых других действий, например как имени целого, некорректно:

```
int Account = 5;          // некорректное использование имени Account
```

Тип Index имеет область действия функции и известен от места его определения до закрывающей фигурной скобки функции main(). Переменные типа Index могут определяться в функции main(), но не в другой области действия, например в функции getBalance():

```
double getBalance(Account a)
{ Index z;                // синтаксическая ошибка: имя Index здесь неизвестно
  return a.bal; }
```

Функция getBalance() действует в масштабе программы. Никакие другие объекты в области действия программы не могут называться getBalance.

Лексическая область действия имен переменных более разнообразна. Переменные C++ могут определяться следующим образом:

- **Переменные блока**. Определяются после открывающей фигурной скобки блока (или в теле блока) и видимы с точки определения до конца блока. В листинге 6.1 переменные блока представляют собой массивы amounts[] и num[], определенные в первом цикле функции main(), переменная temp определяется во втором цикле main(), и снова переменная temp — в третьем цикле main().

- **Переменные функции**. Аналогичны переменным блока, но их область действия — это именованная функция, а не неименованный блок. Они определяются в теле функции (после открывающей фигурной скобки или внутри функции) и видимы от точки определения до закрывающей фигурной скобки функции. В листинге 6.1 переменными функциями являются `i`, `count`, `MAX`, `a[]` и `total`, определенные в функции `main()`, и переменная `total`, определенная в `getBalance()`.
- **Формальные параметры функции**. Определены в заголовке функции и видимы в любом месте в теле функции. Это означает, что имя параметра может конфликтовать с определенной в данной функции переменой. У функции `getBalance()` из листинга 6.1 только один формальный параметр — `a`.
- **Глобальные переменные**. Действуют в масштабе файла. Они определяются в файле вне любой функции и действительны с точки определения до конца файла. В листинге 6.1 нет глобальных переменных. О них рассказывается в следующем примере.

Имена полей структуры локальны для блока определения структуры. Это означает, что на них можно ссылаться (без уточнения) вне данной области действия. В листинге 6.1 имена полей `num` и `bal` известны только в определении структуры `Account`. Следовательно, написать `bal = 0;` в функции `main()` будет некорректно, поскольку имя `bal` в `main()` неизвестно. Но с помощью селектора-точки на эти поля можно ссылаться во всей области действия переменных типа `Account` (где эти переменные известны и видимы). В листинге 6.1 это область действия функции `main()` (где определен массив `a[]` типа `Account`) и область действия функции `getBalance()` (где определен параметр типа `Account`). Поскольку C++ позволяет программистам определять переменные в любом месте области действия, важно убедиться, что имя не используется в области действия перед определением. В листинге 6.1 константа `MAX` должна лексически предшествовать определению массива `a[]`, `amounts[]` и `num[]` в функции `main()`.

Использование одинаковых имен в независимых областях действия

Когда имена определяются в разных областях действия, они не конфликтуют друг с другом (за некоторым исключением).

Понятие "разные" требует в этом случае некоторых пояснений. Как эти области действия должны соотноситься друг с другом, чтобы одно и то же имя могло использоваться в каждой из них с разными целями?

Два блока с непересекающимися областями действия (не имеющие общих операторов) считаются разными и независимыми друг от друга. Например, два неименованных блока, которые прямо или косвенно следуют друг за другом в файле или в области действия, независимы. Они позволяют определять и использовать одно и то же имя для совершенно разных целей. Имена, определенные в независимых областях действия, не конфликтуют друг с другом.

В листинге 6.1 имя `temp` используется в двух циклах функции `main()`. На самом деле нет никакой необходимости применять в этих циклах локальные переменные. Поля элементов массива не могут отображаться непосредственно. Между тем, применение этих переменных хорошо иллюстрирует концепцию области действия. Поскольку каждый из циклов имеет свой набор фигурных скобок, имя `temp`, которое ссылается на разные переменные, не приводит к конфликтам. Их использование не требует координации.

Это же относится к блокам функции, определяющим переменные или параметры с помощью одного имени. Например, переменная `total` определяется и в функции `getBalance()`, и в `main()`. Опять же, в функции `getBalance()` это можно было бы сделать с помощью локальной переменной, но такой пример хорошо иллюстрирует принцип области действия.

Аналогично имя a используется как параметр в функции getBalance() и как массив в функции main(). Когда имена определяются в независимых областях действия, каждое из них известно в своей области, и в координации нет необходимости.

Использование одинаковых имен во вложенных областях действия

Еще один тип разных областей действия — вложенные области. C++ — язык с блочной структурой. Это означает, что лексически его области действия могут быть вложенными, т. е. фигурные скобки одной области действия целиком находятся в другой области действия. Заметим, что разные области действия либо независимы (одна начинается перед другой), либо вложены (одна внутри другой), но не пересекаются.

В большинстве программ C++ используются вложенные области действия. Неименованный блок может быть вложенным в другой неименованный блок или функцию. Нельзя вкладывать неименованный блок непосредственно в область действия файла, так как управление не будет передаваться в этот блок (требуется заголовок функции). Функция может быть вложенной только в области действия файла, но не в другой функции. Например, ниже делается попытка скрыть функцию getBalance() внутри main() — ее имя не будет конфликтовать с другими именами getBalance. Однако это ничего не дает: такие конструкции в C++ не допускаются.

```
int main()
{ double getBalance(Account a)                    // не допускается в C++
  { double total = a.bal;
    return total; }
  ....
  for (i = 0; i < count; i++)
  { total += getBalance(a[i]); }                  // накопление total
  cout << endl << "Total of balances $" << total << endl;
  return 0; }
```

В листинге 6.1 тело цикла реализовано как неименованные блоки. Они вложены в область действия функции main(). Области действия функций main() и getBalance() вложены в область действия исходного файла. По существу, область действия файла вложена в область действия программы.

Введение вложенных областей действия не изменяет правил видимости переменных или типов, определенных во внешней области действия. Они видимы во вложенных областях. Например, переменная count известна от места ее определения до конца функции main(), независимо от того, содержит ли функция main() вложенные циклы. Следовательно, когда неименованный вложенный блок в первом цикле main() ссылается на переменную count, то это ссылка на переменную, определенную во внешнем блоке. Аналогично ссылки на элементы массива a[] имеются во вложенных блоках во всех трех циклах. Переменная total определена в функции main(), а ссылки на нее присутствуют во вложенных блоках третьего цикла.

Не допускается ссылка из внешней области действия на переменные, определенные во вложенной области действия. Например, массивы num[] и amounts[] определены в блоке первого цикла функции main() и не могут использоваться в функции main() вне этого блока. Было бы некорректно написать второй цикл в листинге 6.1 следующим образом, ссылаясь на num[] во внешнем цикле:

```
for (i = 0; i < count; i++)
  cout << num[i] << endl;                  // num[] неизвестно
```

C++ позволяет определять во вложенных циклах переменную, имя которой определено также во внешней, охватывающей области действия. Это обеспечивает взаимодействие имен, определенных во вложенных областях действия. В данном случае программный элемент, определенный во внешней области действия, станет недоступным во вложенной области. Когда имя используется внутри вложенной области действия, оно ссылается на программный элемент, определенный в данной вложенной области. Вне вложенной области действия это имя могло бы тем не менее ссылаться на программный элемент (переменную, тип или параметр), определенный во внешней области действия.

Чтобы продемонстрировать эффект вложенности, рассмотрим листинг 6.2, в нем показана модифицированная версия программного кода, представленного в листинге 6.1. Обе локальные переменные `temp` определены в теле функции `main()` и функции `getBalance()`. Другие бесполезные изменения также сделаны для примера: переменные `MAX` (фактически, это константа), `count` и массив `Account a[]` стали глобальными в области действия файла, а функция `printAccounts()` была добавлена для вывода номера счета и остатка на счете (на отдельной строке) в массиве `a[]`. Сумма остатков на счетах выводится на экран, а затем программа ищет конкретный номер счета и выводит остаток средств на этом счете (если находит его). Результат выполнения программы представлен на рис. 6.2.

```
Данные загружены

800123456    1200
800123123    1500
800123333    1800

Остаток на счете в долл. 4500
На счете 800123123 осталось 1500
```

Рис. 6.2. *Вывод программы из листинга 6.2*

Листинг 6.2. Демонстрация вложенных областей действия и наложения имен

```
#include <iostream>
using namespace std;

struct Account {
   long num;
   double bal; } ;

const int MAX = 5;                                  // максимальный размер набора данных
int count = 0;                                      // число элементов в наборе данных
Account a[MAX];                                     // глобальные данные для обработки

void printAccounts()
{ for (int i = 0; i < count; i++)                   // глобальный счетчик
   { double count = a[i].bal;                       // локальный счетчик
     cout << a[i].num << " " << count << endl; } }

int main()
{
   typedef int Index;
   long num[MAX] = { 800123456, 800123123, 800123333, -1 } ;
   long number = 800123123; double total = 0;       // внешняя область действия
   while (true)                                     // конец, если контрольное значение
   { double amounts[MAX] = { 1200, 1500, 1800 } ;   // данные для загрузки
     if (num[count] == -1) break;                   // найдено контрольное значение
   double number = amounts[count];                  // number скрывает внешнее имя number
   a[count].num = num[count];                       // загрузка данных
   a[count].bal = number;
   count++; }
   cout << " Данные загружены\n\n";
   printAccounts();
   for (Index i = 0; i < count; i++)                // глобальный счетчик
     { double count = a[i].bal;
       total += count;                              // локальный счетчик
```

```
    if (i == ::count - 1)                          // глобальный счетчик
       cout << "Остаток на счете в долл." << total << endl; }
  for (Index j = 0; j < count; j++)
    if (a[j].num == number)
  cout << "На счете " << number << " осталось: долл. " << a[j].bal << endl;
  return 0;
}
```

Областью действия глобальных переменных будет файл, в котором они определены. Любая функция в данном файле может ссылаться на такое имя (если оно не скрытое), и все эти ссылки будут указывать на одну глобальную переменную. Например, массив `a[]` и переменная `count` в листинге 6.2 ссылаются только на функцию `printAccounts()`, а константа `MAX` в `main()` используется только в функции `main()`. Чтобы работать с этими именами в `printAccounts()`, нет никакой необходимости определять их там. Достаточно глобальных определений.

По существу областью действия глобальных переменных является программа, а не файл. Если определить имя `MAX`, `count`, `a` или `num` как глобальное имя в другом файле той же программы, то каждый файл компилируется отдельно. При компиляции не проверяется содержимое других файлов. Независимо от того, для чего используются имена — для одной цели или разных, компоновщик сообщит о дублированных определениях. Например, `a[]` и `num[]` могут определяться в другом файле как скалярные переменные, а не массивы. Их повторное использование приведет к ошибке. Это относится только к глобальным определениям и не применяется ни к объявлениям, ни к неглобальным определениям. Позднее будут приведены примеры.

Другие области действия C++ (функции или блока), определенные в конкретном исходном файле, будут вложенными в глобальную область действия файла. Следовательно, глобальные имена видимы в функциях внутри файла, как любые внешние имена во вложенных циклах. Если функции сами содержат вложенные области действия, то имена глобальных переменных будут видимы и в этих областях. В листинге 6.2 глобальные массивы `a[]`, `num[]` и индекс `count` используются в теле первого цикла, вложенного в область действия `main()`. Наличие вложенных (на любую глубину) областей действия не изменяет видимости имен, определенных во внешних областях действия.

Во вложенных областях действия переменные могут определяться с помощью имен из внешних областей действия (которые уже известны во вложенных областях). Когда такое имя используется в локальной вложенной области действия (функции в файле, в расположенном в функции блоке или в другом блоке), то данная ссылка указывает на локальное имя. Если такое имя используется во внешней, охватывающей области действия, ее смысл определяется этой внешней областью (локальное имя вне конкретной области действия неизвестно).

В листинге 6.2 функция `printAccounts()` использует в цикле с условием продолжения имя `count`. Это имя ссылается на глобальную переменную `count`. Внутри цикла имя `count` ссылается на переменную, определенную в теле цикла, а не в глобальной области действия. Имя во вложенной области действия переопределяет глобальное имя. Иногда говорят не о переопределении имен во вложенных областях, а о сокрытии имени. Заметим, что имя во вложенной области необязательно определяет переменную того же типа. Это может быть все, что угодно.

Нетрудно написать функцию `printAccounts()` без использования переменной `count`. Эта переменная была представлена только для иллюстрации на относительно простом примере концепции области действия имен. Фактически невозможно предложить пример, где повторное использование глобального имени было бы действительно необходимо. Можно всегда обойтись локальным именем, отличным от имени во внешней области действия. Вся прелесть принципа области действия имен в том, что не требуется определять другое имя. Можно использовать то имя,

которое вам нравится, и это имя будет известно в данной области действия, независимо от того, какие имена действуют во внешней области.

Когда во вложенной области действия переопределяется имя, определенное во внешней, охватывающей области (глобальной или вложенной в другую область), имя из внешней области действия становится недоступным во вложенной области. Переопределение имени из внешней области указывает сопровождающему приложение программисту, что в намерения разработчика не входило использование глобального имени в локальной области.

В листинге 6.2 тело первого цикла в функции `main()` определяет переменную `number`. Такое же имя `number` определяется в области действия самой функции `main()`. Это означает, что в теле цикла `number` действует локальная переменная типа `double`, а не внешняя типа `int`, так как локальное имя переопределяет внешнее. Вне цикла (например, на предпоследней строке в листинге 6.2) имя `number` ссылается на переменную, определенную в самой функции `main()`.

Аналогично в теле второго цикла функции `main()` в листинге 6.2 определяется переменная `count` типа `double`, переопределяющая глобальную переменную `count` типа `int`. Ссылки на имя `count` в цикле разрешаются компилятором как ссылки на локальную переменную типа `double`, хотя в условии продолжения цикла это ссылка на глобальную переменную `count` типа `int`.

Если во вложенной области действия требуется доступ к глобальному имени, можно использовать операцию C++ глобальной области действия '`::`'. Например, в листинге 6.2 сумма остатков на счетах выводится внутри, а не вне цикла (что было бы проще и естественней). Таким образом, в цикле нужно сравнить индекс `i` с числом допустимых элементов набора данных. В таком контексте `::count` во втором цикле `main()` ссылается на глобальный объект `count`, а не на локальный объект.

Понятно, что доступ к скрытым глобальным объектам не должен быть простым. Если во вложенной области действия требуется глобальное имя, его нельзя переопределять. Кроме того, чтобы избежать конфликта имен, во вложенной области действия можно ввести любое имя. Операция области действия может понадобиться при сопровождении программы, когда возникают новые требования к применению уже переопределенного глобального имени, а в первоначальном варианте программы такое использование не предусматривалось.



Осторожно! Операция глобальной области действия `::` переопределяет правила действия областей. Для сопровождающего программиста легче придерживаться правил области действия, чем искать эту операцию. Присваивайте имена глобальным переменным так, чтобы свести к минимуму необходимость применения данной операции.

Обратите внимание, что операция области действия позволяет обращаться только к глобальным переменным. В программе C++ нет механизма, с помощью которого можно из вложенной области действия обращаться к переменным внешней области, переопределенным во вложенной.

В листинге 6.2 в теле первого цикла определяется переменная `number`, скрывающая переменную `number`, определенную в `main()`. Это означает, что все ссылки на `number` в данном цикле являются ссылками на локальную переменную. К определяемой в `main()` переменной `number` можно обращаться вне тела цикла (например, в последнем цикле из листинга 6.2).



Внимание Операция области действия позволяет обращаться к глобальному имени. Если во вложенном цикле переопределяется имя, действующее во внешнем блоке, то C++ не предусматривает способа ссылки на это внешнее имя. Если во вложенном блоке действительно необходимо такое внешнее имя, просто не переопределяйте его.

Область действия переменных цикла

Определение переменных в заголовке цикла позаимствовано из аналогичных средств языка Ада, но в C++ оно реализуется по-другому, и разными компиляторами интерпретируется по-разному. Когда имя переменной цикла используется вне цикла, одни компиляторы сигнализируют об ошибке, другие — нет. Новый стандарт C++ ограничивает область действия переменных цикла телом цикла, следовательно, они не должны использоваться вне цикла. Если в другом цикле в данной области действия то же имя используется для другой переменной цикла, некоторые компиляторы указывают на ошибку (хотя новый стандарт это допускает), другие ее игнорируют. В листинге 6.2 показаны примеры предусмотрительного использования данных средств с учетом переносимости программы: переменные цикла не переопределяют имен из внешних областей действия, не используются вне циклов и не переопределяются в других циклах в той же области действия.

В общем случае лексическая область действия является важным средством. Имена могут просто использоваться повторно (без конфликтов) в независимых областях действия и переопределяться (путем сокрытия внешних имен) во вложенных областях. Если области действия объектов с одним и тем же именем являются вложенными, то последние определенные имена скрывают предыдущие.

Правила области действия имен помогают избегать конфликтов имен и излишней координации между программистами, работающими над одним проектом.

Управление памятью: классы памяти

Лексическая область действия, о которой рассказывалось в предыдущем разделе, является характеристикой программы этапа компиляции. Она определяет сегменты исходного кода программы, где известно конкретное имя. Однако лексическая область действия не определяет, когда на этапе выполнения программы выделяется память для конкретной переменной и когда эта память освобождается для других целей. Правила распределения памяти на этапе выполнения зависят от другой характеристики определяемых в программе имен — их классов памяти (или экстента).

Классом памяти называется интервал выполнения программы, на котором действует связь между именем переменной и ее адресом в памяти, т. е. когда память выделена для данной переменной. В отличие от лексической области действия, класс памяти определяет поведение программы на этапе выполнения.

Выполнение программы в C++ всегда начинается с функции `main()`. Первый оператор в функции `main()` обычно является первым выполняемым программой оператором. Функция `main()` вызывает другие функции программы, которые, в свою очередь, вызывают функции. Когда функция-сервер завершает свою работу (выполняет оператор `return` или достигает закрывающей фигурной скобки в теле функции), управление возвращается к вызвавшей ее клиентской функции. Когда последняя вызванная из `main()` функция завершается и функция `main()` достигает закрывающей фигурной скобки (или оператора `return`), завершается вся программа.

До сих пор рассматривались две версии функции `main()`: одна с возвращаемым типом `int`, а другая — возвращающая `void`. Если тип не указывается, компилятор предполагает, что это функция `int` (что, конечно, не всегда уместно). Каждая форма функции `main()` может использоваться с необязательными параметрами:

```
void main(int argc, char* argv[])              // аргументы командной строки
{ for (int i = 0; i < argc; i++)               // начало выполнения программы
  cout << "Аргумент " << i << ": " << argv[i] << endl;
.... }                                         // конец программы
```

Параметры передаются функции main() из операционной системы при вызове программы. Они содержат аргументы командной строки, введенные пользователем при вызове программы (если они есть). Эти параметры определены как счетчик аргументов командной строки (argc) и массив (вектор) строк (argv[]), где каждая строка содержит один из аргументов (ниже рассказывается об указателях).

Часто эти строки представляют собой имена файлов, набранных в командной строке. В приведенном выше примере функция main() использует count аргументов командной строки и анализирует каждый из аргументов (в данном случае — просто выводит его на экран). В список аргументов командной строки включено имя выполняемого файла программы. Конечно, строка с именем файла программы имеет в массиве нулевой индекс. Например, если выполняемый файл программы называется copy, то команда:

```
c:\>copy account.ccp c:\data
```

дает следующие строки:

```
Agrument 0: copy
Agrument 1: account.cpp
Agrument 2: c:\data
```

В ходе выполнения программы память для ее переменных (объектов) может выделяться в трех зарезервированных для программы областях: в фиксированной памяти, области стека и динамически распределяемой области. Пока что мы не будем рассматривать, как происходит управление этими областями памяти в конкретной компьютерной архитектуре. Какой бы ни была переменная (скалярной, массивом, структурой, классом, объединением или перечислением), память для нее выделяется в одной из данных областей при выполнении программы (в зависимости от ее класса памяти).

Концепция класса памяти заставляет внести некоторые дополнения в принцип области действия имен. Переменные, определенные как глобальные в области действия файла, помещаются в фиксированную область памяти. Переменные, определенные как локальные для функции или блока, помещаются в стек. Кроме того, C++ поддерживает динамические переменные. Они не определяются как глобальные или локальные и, следовательно, не имеют имен. Память для них выделяется в динамически распределяемой области с помощью явных операций (операции new).

В определениях переменных классы памяти C++ можно задавать с помощью следующих ключевых слов:

- auto: назначается по умолчанию для переменных, определяемых как локальные для блока или функции
- extern: может применяться к переменным, глобальным в области действия файла
- static: может использоваться для глобальных переменных в области действия файла или для локальных переменных, определенных в области действия блока или функции
- register: используется для переменных, хранимых в быстродействующих регистрах, а не в памяти с произвольным доступом

Для объектов (переменных) этих классов правила языка определяют механизмы выделения и освобождения памяти: переменные extern и static распределяются в фиксированной области памяти программы, переменные auto — в стеке, а переменные register — в регистрах (если это возможно). Если нет доступных регистров, то данные переменные распределяются в фиксированной области (в случае глобальных переменных) или в стеке программы (если переменные локальные).

Автоматические переменные

Автоматические переменные — это локальные переменные, определенные в блоках или функциях. Спецификатор auto задается по умолчанию и используется редко. Например, функцию printAccounts() в листинге 6.2 можно записать следующим образом:

```
void printAccounts()
{ for (auto int i = 0; i < count; i++)          // глобальная переменная count
  { auto double count = getBalance(a[i]);       // локальная переменная count
    cout << a[i].num << " " << count << endl; } }
```

Поскольку программистам обычно не нравится набирать лишнее, они предпочитают опускать спецификаторы по умолчанию, если нет явных причин их указывать.

Память для автоматических переменных выделяется из стека, когда программа достигает открывающей фигурной скобки функции или блока. Если в определение включена инициализация, как в примере с printAccounts(), выделяемая для переменной память инициализируется. Если начальное значение в определении не задается, значение переменной не определено. Скорее всего, она будет содержать значение, оставшееся в данной области памяти от предыдущей переменной. В C++ нет ключевого слова "undefined" (не определено), но его следует принимать всерьез. Если необходимо конкретное значение, инициализируйте переменную и используйте его, но на неопределенные значения полагаться не следует. Они могут быть совершенно произвольными и различаться при каждом запуске программы, даже когда эксперименты показывают, что они одни и те же. Не доверяйте подобным экспериментам.

При выполнении программы автоматические объекты существуют в памяти только после входа в область действия, где они определены. Они выделяются в стеке программы (и на них можно ссылаться по имени) и существуют, пока программа не доходит до закрывающей фигурной скобки области действия. В этот момент память возвращается в стек и может использоваться для других целей.

C++ предоставляет замечательные возможности управления памятью. Они освобождают программиста от обязанностей распределения памяти для отдельных объектов. Для некоторых задач этих методов недостаточно и вместо них применяется динамическое распределение памяти. Как будет показано в данной главе, динамическое распределение памяти — вещь более сложная и способствующая появлению ошибок. Именно поэтому как можно чаще следует использовать автоматические переменные.

Память, распределяемая для автоматической переменной при другом вызове той же функции (или на очередной итерации того же цикла), может оказаться в другом месте стека и с другим содержимым, поэтому автоматические переменные нельзя использовать для передачи данных между последовательными вызовами функции или последовательными итерациями цикла. Если переменная не инициализируется, она будет иметь на каждой итерации неопределенное значение. Если значение переменной включает инициализацию, инициализация повторяется при каждом входе в область действия. В примере с printAccounts() память для локальной переменной count выделяется, инициализируется и освобождается на каждой итерации цикла. Кроме того, память выделяется, инициализируется и освобождается при каждом обращении к функции printAccounts().

Когда памяти для программы достаточно и скорость ее выполнения не критична, не следует пытаться оптимизировать управление памятью для локальных переменных. Если же ресурсов не хватает, важно хорошо понимать последствия выбора архитектуры программы. Например, в листинге 6.2 массив num[] определяется как локальная переменная в функции main(), а массив amounts[] — как локальная переменная в теле первого цикла. Оба этих массива содержат данные

для загрузки значений в глобальный массив a[]. Определение массивов num[] и amounts[] в разных местах программы — пример разделения того, что должно быть вместе.

Данное решение может также повлиять на производительность. Память для массива num[] выделяется только один раз, в начале выполнения функции main(). Память для массива amounts[] выделяется, инициализируется и освобождается столько раз, сколько выполняется тело цикла. Выделение и освобождение памяти не занимает много времени при выполнении программы (это операции с указателем стека), но копирование значений в элементы массива при инициализации — процесс более длительный. Он требует почти столько же времени, сколько копирование данных из массива amounts[] в массив a[]. Было бы хорошо выделить память для массивов num[] и amounts[] в одном месте и делать это только один раз при выполнении программы.

```
int main()
{ typedef int Index;
  long num[MAX] = { 800123456, 800123123, 800123333, -1 };
  double amounts[MAX] = { 1200, 1500, 1800 } ;          // данные для загрузки
  long number 800123123; double total = 0;              // внешний цикл
  while (true)
  { if (num[count] == -1) break;
    ...,..} }                                           // конец main()
```

Имена автоматических переменных невидимы вне их области действия, поэтому их можно повторно задавать в других областях действия. Связь между адресами памяти для переменных и их именами в различных областях действия отсутствует. Для уменьшения координации между разработчиками это хорошо. Когда в разных областях действия используется глобальная переменная, в каждой области действия ссылка идет на один и тот же адрес памяти, поэтому нужно понять, действительно ли можно в разных целях задействовать одну ячейку памяти или нужно ввести разные переменные. Применение автоматических переменных упрощает задачу разработчика и программиста, сопровождающего программу.

Согласно правилам области действия, во вложенных блоках имя можно повторно использовать для других объектов. Память для нового объекта с тем же именем выделяется в стеке в ячейке, отличной от ячейки, выделенной для переменной во внешнем цикле. Имя во вложенной области действия скрывает объект, распределенный в стеке ранее (и все еще существующий). Например, в листинге 6.2 переменная number определяется в функции main() и переопределяется в теле первого цикла функции main(). Вторая переменная number распределяется в стеке в начале итерации цикла и освобождается в конце каждой итерации. Это совершенно разная память (даже на разных итерациях), и она не имеет ничего общего с памятью, выделенной для переменной number в начале функции main(). Поэтому, когда на третьем цикле в main() требуется значение, присвоенное переменной number в начале функции main(), после завершения первого вложенного цикла его можно использовать снова — оно осталось неприкосновенным.

Аналогично, когда функция main() вызывает функцию printAccounts(), для каждой итерации в printAccounts() в стеке выделяется память для переменной count. Эти области памяти на каждой итерации могут быть разными и не имеют ничего общего с глобальной переменной count в фиксированной области данных.

Если вложенная область памяти не скрывает переменной, определенной во внешней области, имя этой переменной доступно во вложенной области действия. В листинге 6.2 память для переменной total выделяется в начале функции main() и не переопределяется во вложенных областях действия. Когда второй цикл ссылается на total, это будет ссылка на переменную, определенную во внешней области действия.

Формальные параметры функции интерпретируются как автоматические переменные, определенные в области действия функции. Они инициализируются значениями фактических аргументов в вызове функции. Например, в первой версии примера программы (в листинге 6.1) функция `getBalance()` инициализирует свой параметр значением `a[i]` в функции `main()`. Память для параметров выделяется в стеке, когда начинается выполнение функции, а освобождается, когда при выполнении функции достигается закрывающая фигурная скобка.

В общем случае лучше определять переменную по возможности на самом глубоком уровне вложенности в структуре блоков. Это дает следующие преимущества:

- Минимизируется область действия программы, где известно имя, а следовательно, и потенциальные конфликты имен с другими объектами
- Память увязывается с переменной в течение минимального интервала времени. В остальное время эту память можно использовать для других целей

Нужно также продумать вопросы доступности объекта в других частях программы и отрицательное влияние на производительность программы из-за повторного выделения памяти, ее инициализации и освобождения. Необходимо принимать во внимание и опасность нехватки памяти в стеке. Общие потребности в памяти зависят от последовательности вызовов функций, и ни компилятор, ни программист не могут точно предсказать ее. Это особенно важно при определении массивов как локальных переменных в функциях и во вложенных блоках, например массива `amounts[]` в листинге 6.2.

Внешние переменные

Внешние, или глобальные, переменные — это переменные, определяемые вне любой функции. Как уже упоминалось, их область действия — файл (от точки определения до конца файла). Следовательно, в другом файле нельзя ссылаться на ту же переменную — это имя будет там невидимо. (На самом деле данное препятствие легко преодолеть.) В то же время такое имя можно использовать в другом файле для определения другой внешней переменной. По существу, имена глобальных переменных действуют в масштабе программы (аналогично именам функций C++).

Память для глобальных переменных распределяется не так, как для автоматических. Она выделяется из фиксированной области данных в начале выполнения программы, непосредственно перед первым оператором функции `main()`. Адрес памяти ассоциируется с именем переменной до завершения программы и освобождения памяти после выполнения завершающего оператора `main()`.

В определениях глобальных переменных допускается инициализация. Если инициализация отсутствует, переменная инициализируется нулем соответствующего типа. Это важное отличие от автоматических переменных, которые не имеют значений по умолчанию — их начальное значение не определено (программисты часто называют его "мусором").

В листинге 6.2 переменные `MAX`, `count` и `a[]` определяются как глобальные.

Легко подсчитать общий объем памяти для всех глобальных переменных в программе. Компилятор обрабатывает каждый файл отдельно и вычисляет, сколько пространства необходимо для глобальных переменных, складывая размеры всех глобальных объектов. (Для автоматических переменных это не имеет смысла, поскольку они не существуют в памяти одновременно.) Еще одно преимущество использования глобальных переменных — скорость: поскольку память для глобальной переменной выделяется и освобождается только один раз, а не при каждом вхождении в область действия, такая операция не замедляет выполнения программы (хотя для многих приложений это не важно).

Другое преимущество применения глобальных переменных состоит в меньших требованиях к стеку программы. Необходимый программе размер стека невозможно вычислить точно, а потому всегда есть вероятность нехватки стека. Вот почему важно не увеличивать требований к стеку без веской причины. Например, массив amounts[] в листинге 6.2 определяется как локальный, а массив num[] — как глобальный. Здесь не только разделено то, что должно быть вместе, и не только тратится время на выделение памяти и инициализацию массива на каждой итерации цикла. Для массива amounts[] отводится память в стеке. Первые две операции требуют времени, а третья — дополнительной памяти. Если сделать данный массив глобальной переменной, все три недостатка будут устранены. В данном примере массив содержит только три переменные — и стека вполне хватит, однако многие программисты выделяют в стеке память для больших массивов, не понимая всех последствий.

Еще одно преимущество, по крайней мере для некоторых программистов, состоит в возможности обойтись за счет глобальных переменных без параметров функций. Так как областью действия глобальной переменной является весь файл, где она определена, любая функция, определенная в том же файле после глобальной переменной, может обращаться непосредственно к этой переменной. Например, функция printAccounts() в листинге 6.2 прямо обращается к глобальной переменной count и a[] без сложностей (и дополнительного времени) передачи параметров. Другие полагают, что прямой доступ к глобальным переменным затрудняет понимание интерфейса функции сопровождающим программистом. Чтобы понять, какие переменные функция использует и модифицирует, нужно проверить каждую строку исходного кода функции. Как будет показано ниже, параметры можно документировать непосредственно в интерфейсе функции — не потребуется проверки каждой строки кода.

Недостатком глобальной переменной является длительное время существования. Она функционирует в течение всего времени выполнения программы, что затрудняет использование памяти для других целей. Например, в листинге 6.2 переменные count и a[] действуют во всей программе. С другой стороны, массивы num[] и amounts[] необходимы только в теле первого цикла main(), после чего занятую ими память можно освободить для других целей. Именно это происходит с массивом amounts[], для которого отводится память в стеке, однако массив num[] сохраняется. Его повторное использование требует аккуратного планирования при разработке программы и может значительно затруднить ее сопровождение. Вот почему в данной программе все переменные не определялись как глобальные.

Имя переменной, определенной в исходном коде как глобальная, известно в любой области действия, вложенной в данный файл. Можно обращаться к глобальной переменной из любого места в файле. Например, в листинге 6.2 переменная count используется в main() как счетчик цикла, MAX применяется для определения массивов a[], num[] и amounts[]. На глобальный массив num[] имеется ссылка в main(), а на глобальный массив a[] — в функции main() и в функции printAccounts().

Как уже упоминалось, во вложенной области действия глобальное имя может переопределяться (скрываться). Память при таком переопределении берется из стека, а не из фиксированной области памяти, и имя будет ссылаться на локальную автоматическую переменную, а не на глобальную. В листинге 6.2 функция printAccounts(), как и второй цикл в main(), используют имя count для автоматической переменной. Когда к автоматической переменной применяется операция области действия '::', эта переменная ссылается на ячейку памяти в фиксированной области, а не на ячейку памяти в стеке (::count в листинге 6.2).

Если локальная переменная count определяется в одной из функций в другом файле программы из листинга 6.2, то это не приведет к проблемам, так как области действия независимы. Такая функция будет ссылаться на ячейку памяти в стеке. Если же в другом файле определяется глобальная переменная count (а это популярное имя — короткое и выразительное), то возникнут ошибки при компоновке.

Применение глобальных переменных требует дополнительной координации и согласования имен между программистами, работающими над разными файлами программы.

В то же время на глобальную переменную, определенную в одном файле, можно ссылаться из других файлов приложения, и это еще одна причина в пользу применения глобальных переменных.

Чтобы переменная, определенная в одном файле, была известна в другом файле, используется ключевое слово extern. Это означает не повторное использование имени глобальной переменной для других целей, а ссылку на ту же область памяти с помощью того же имени.

Предположим, что программа из листинга 6.2 претерпела дальнейшее развитие и разбита на большее число функций и эти функции должны быть размещены в разных файлах, чтобы с ними могли работать другие программисты. Допустим, вместо поиска конкретного номера счета в конце функции main() нужно вызвать функцию printAverage(), которая использует вычисляемую в main() сумму остатков на счетах, определяет среднее и выводит его на экран. Вместо применения в операторе cout литерального значения используется переменная caption[], содержащая текст "Средний остаток на счетах в долл." (распространенный метод, облегчающий локализацию программы). Кроме того, функция printAverage() вызывает функцию printCaption(), использующую переменную caption[]. Этот простой пример достаточно легко понять, но введение в нем дополнительных функций требует обсудить вопросы, имеющие важное значение при разработке больших программ.

Для реализации функций printAverage() и printCaption() в другом исходном файле потребуются:

- Исходный файл, вызывающий функцию printAverage(), т. е. файл с функцией main(), где известно, что printAverage — имя функции, определенной в другом файле
- В файле, где реализованы функции printAverage() и printCaption(), должны быть известны глобальные переменные count и caption[], определенные в другом файле

```
   Данные загружены

800123456    1200
800123123    1500
800123333    1800

   Данные обработаны

Средний остаток на счетах в долл. 1500
```

Рис. 6.3. *Вывод программы из листинга 6.3 и листинга 6.4*

В листинге 6.3 приведена модифицированная программа из листинга 6.2, решающая эти проблемы. Функция printAccounts() упрощена, устранен тип Index, массив amounts[] определяется вслед за массивом num[] (как уже говорилось, эти объекты должны быть вместе), функция printAverage() вызывается в конце функции main(). Добавлен глобальный массив caption[] с заголовком для печати среднего остатка на счетах. В листинге 6.4 показан второй файл, где реализованы функции printAverage() и printCaption(). Вывод программы представлен на рис. 6.3.

Как видно в листинге 6.3, первая проблема решается добавлением к исходному файлу прототипа функции printAverage() с предшествующим ключевым словом extern:

```
extern void printAverage(double);       // определяется в другом месте
```

Если ключевое слово extern в объявлении функции опущено, компилятор и компоновщик все равно будут искать где-либо эту функцию. Некоторые программисты предпочитают использовать в программах C++ ключевое слово extern, чтобы избежать проблем переносимости:

```
void printAverage(double);              // все равно определяется в другом месте
```

Листинг 6.3. Коммуникации с другим файлом через внешние описания (часть 1)

```
#include <iostream>
using namespace std;

struct Account {                                          // глобальное определение типа
   long num;
   double bal; } ;

extern void printAverage(double total);                   // определяется где-то еще

const int MAX = 5;
Account a[MAX];                                           // глобальные данные для обработки
int count = 0;                                            // число элементов в наборе данных
char caption[] = "Средний остаток на счетах в долл. ";    // заголовок для печати

long num[MAX] = { 800123456, 800123123, 800123333, -1 } ;
double amounts[MAX] = { 1200, 1500, 1800 } ;              // набор данных для загрузки

void printAccounts()
{ for (int i = 0; i < count; i++)                         // глобальный счетчик
  cout << a[i].num << " " << a[i].bal << endl; }

int main()
{
   double total = 0;
   while (true)                                           // выход по контрольному значению
   { if (num[count] == -1) break;
     a[count].num = num[count];                           // глобальные a[], num[], amounts[]
     a[count].bal = amounts[count++]; }                   // загрузка данных
   cout << " Данные загружены\n\n";
   printAccounts();                                       // локальная функция
   cout << "\n Данные обработаны\n\n";
   for (int i = 0; i < count; i++)
     { total += a[i].bal; }
   printAverage(total);                                   // глобальная в другом файле
   return 0;
}
```

В применении к переменным ключевое слово extern имеет другой смысл. Во-первых, оно означает, что определенная в данном файле локальная переменная может быть доступна в другом файле; во-вторых, обозначает переменную, определенную в другом файле и объявленную в этом файле, чтобы ее могли видеть содержащиеся в файле функции. В первом случае применение ключевого слова extern необязательно, во втором — обязательно.

Кажется сложным? Не беспокойтесь — все достаточно просто. Ключевое слово extern необязательно в определениях и обязательно в объявлениях. Рассмотрим примеры внешних переменных из листинга 6.3. Глобальные переменные из листинга 6.3 перечислены в определениях, следовательно, они являются внешними неявно и видимы в других файлах. Использовать ключевое слово extern нет необходимости, но, если оно указывается и переменная инициализируется, ничего плохого не случится.

```
extern int count = 0;                    // OK: это определение
```

Наличие инициализации указывает компилятору, что это определение, а не объявление. Если инициализация отсутствует, то без нее определение превращается в объявление и компоновщик сообщит об отсутствии определения для count:

```
extern int count;                        // это объявление
```

Отсутствие инициализации и ключевого слова extern снова делает его определением (конечно, значение можно инициализировать в другом месте), и к переменной можно обращаться из другого файла (листинг 6.4, где определяется printAverage[]).

```
int count;                    // OK: это определение
```



Осторожно! Все глобальные переменные являются внешними по умолчанию. Применение ключевого слова extern необязательно. Оно сообщает сопровождающему приложение программисту, что переменная используется в других файлах. Если глобальная переменная не инициализируется при определении, при включении в строку ключевого слова extern компоновщик принимает ее за объявление.

Массив caption[] из листинга 6.3 инициализируется. Следовательно, это определение (память для массива выделяется в фиксированной области), массив является по умолчанию внешним (extern) и может использоваться в другом файле (листинге 6.4, где определяется printCaption[]). Массивы num[] и amounts[] также глобальные — с ними можно работать в других файлах. Но здесь этого не делается (и не должно, так как они просто содержат для программы данные инициализации). То, что caption[] используется в других файлах, а num[] и amounts[] нет, не вполне очевидно для программиста. Скоро мы исправим этот недостаток, введя статический класс памяти static.

В листинге 6.4 приведена функция printCaption(). Она вызывается из этого файла функцией printAverage() и использует массив caption[], определенный в файле из листинга 6.3. Переменная count также определена как extern без инициализации, что превращает определение в объявление. Пропуск ключевого слова сделал бы его определением, но компоновщик будет помечать два определения count как ошибку (даже если их типы разные). Однако компилятор обрабатывает исходные файлы по отдельности и пропускает эту проблему. Применение ключевого слова extern позволяет одному файлу обращаться к данным и функциям, определенным в другом файле, но не сообщает сопровождающему приложение программисту, какие глобальные переменные и функции используются в других файлах (как printAverage()), а какие нет (как printCaption()). Применение ключевого слова static позволяет решить данную проблему.

Листинг 6.4. Коммуникации с другим файлом через внешние описания (часть 2)

```
#include <iostream>
using namespace std;
extern count;                    // определяется и инициализируется в другом месте
extern char caption[];           // определяется и инициализируется в другом месте

void printCaption()              // вызывается только из этого файла
{ cout << caption; }

void printAverage(double sun)    // вызывается из другого файла
{ if (count == 0) return;
  printCaption();
  cout << sum/count << endl;
}
```

Заметим, что объявление массива caption[] не требует указания размера массива, так как при объявлении объекта память не выделяется — оно указывает наличие определения данного объекта в другом месте. Не следует также инициализировать объекты extern: это превратит объявление в определение (и вызовет конфликт имен).

В отличие от определений внешние объявления могут повторяться в других файлах или даже в том же файле. При наличии таких объявлений программный код данного файла может использовать глобальные имена точно так же, как если бы определения содержались в этом файле. Например, в листинге 6.4 функция `printAverage()` ссылается на `count`, а функция `printCaption()` — на переменную `caption[]`, определенную в другом файле (листинг 6.3).

Внешние переменные предоставляют хороший инструмент взаимодействия между функциями в разных файлах большой программы. Их нужно использовать только тогда, когда преимущества распределения функций по разным файлам перевешивают преимущества их включения в один файл. В листингах 6.3 и 6.4 представлен хороший пример избыточных коммуникаций между файлами. Совмещение тех программных элементов, которые должны быть вместе, устраняет необходимость коммуникаций между файлами, описаний `extern`, упрощает проектирование и сопровождение программы, снижает вероятность ошибок.

Статические переменные

Ключевое слово `static` имеет в C++ пять значений. Для статических переменных характерны некоторые общие особенности. (Память для них выделятся в фиксированной памяти, а не в стеке.) Однако различия между значениями слова `static` весьма существенны, и, применяя его в разных контекстах, можно легко запутаться. Как `static` определяются следующие объекты C++:

- Глобальные переменные, которые должны быть доступны только в том же файле, где они определяются, но не в других файлах
- Локальные переменные, определенные в функции (или в неименованном блоке), значения которых должны сохраняться между вызовами функций (или при переходе из одной области выполнения в другую)
- Поля структуры или класса, которые при использовании всех переменных или объектов данного класса должны ссылаться только на одну ячейку памяти
- Функции-члены класса, обращающиеся только к параметрам, глобальным переменным и статическим переменным класса, но не к нестатическим полям класса
- Глобальные функции (не члены класса), доступные для кода клиента только в том же файле, где они определены, но не в других файлах

Пока что рано обсуждать все эти пункты, поэтому мы остановимся на первых двух. Об остальных рассказано в главе 8.

Первый вариант использования ключевого слова `static` — для глобальных переменных — представляет мощное средство, позволяющее создавать закрытые (для файла) переменные. Никакой другой файл не сможет обращаться к таким переменным, объявляя их как `extern`. Например, в листинге 6.3 определяются глобальные переменные `MAX`, `a[]`, `count`, `caption[]`, `num[]` и `amounts[]`, но не указывается, какие переменные доступны из других файлов. Чтобы указать, что переменная `count` доступна из других файлов, и обеспечить недоступность прочих глобальных переменных в других файлах (ведь все функции, обращающиеся к этим глобальным переменным, находятся в том же файле), в листинге 6.3 можно определить эти глобальные переменные как `static`:

```
int count = 0;                  // может быть объявлена как extern в других файлах
static const int MAX=5;         // нельзя сделать extern в другом месте
static Account a[MAX];          // нет доступа из программ в других файлах
static long num[MAX]={ 800123456, 800123123, 800123333, -1 } ;
static double amounts[MAX] = { 1200, 1500, 1800 };
```

При добавлении в определение глобальной переменной слова static не меняется ни место, где выделяется память для переменной, ни время ее существования (от начала до конца выполнения программы). Единственный результат состоит в том, что переменная не может определяться как extern в других исходных файлах и недоступна из других файлов программы. Настоятельно рекомендуется использовать именно такой метод программирования.

Обратите внимание, что массив caption[] отсутствует среди этих глобальных переменных. Поскольку он используется только функцией printCaption[] (в листинге 6.4), его не следует отделять от этой функции и помещать в файл из листинга 6.3, где ни одна функция к нему не обращается. Этот массив лучше переместить в файл из листинга 6.4. Так как функции, определенные в других файлах, не обращаются к данному массиву, он может (и должен) объявляться в листинге 6.4 как static. Начало листинга 6.4 может выглядеть так:

```
extern count;                                  // определяется в другом месте
static char caption[]                          // нет extern, определяется и
                                               // инициализируется здесь
  = "Средний остаток на счетах в долл. ";      // используется локально,
                                               // нет в других файлах
```

Некоторые программисты считают, что это скорее мера безопасности. Применение подобных переменных static исключает ошибки, предотвращая случайные или несанкционированные изменения в других частях программы. Все это так, но подобные ошибки достаточно редки. Реальная ценность данного метода — в сведении к минимуму необходимости координации между разработчиками. Если определять глобальные переменные как static, то можно использовать такие популярные имена как MAX, a, num, amounts и caption в одном файле программы, не координируя выбора имен.

В общем случае использование глобальных переменных лучше ограничивать. Когда они применяются для коммуникаций между функциями в одном файле, то описание переменных как static уменьшает необходимость взаимодействия между программистами, работающими над разными файлами (главное — не разделять те компоненты, которые должны располагаться вместе). Конечно, если глобальная переменная определяется как static, к ней нельзя обращаться из другого файла. Если же не сделать ее статической, то нет гарантии, что к ней действительно не будет обращений из другого файла, или разработчик программы никак не сообщит, что данная переменная используется только в одном файле. Именно поэтому нужно кропотливо указывать ключевое слово static для глобальных статических переменных.

Метод определения статических глобальных переменных очень важен в языке C. Именно так в этом языке был впервые использован объектно-ориентированный подход. Данные и функции связывались в одном файле (как массив caption[] и функция printCaption() после перемещения массива в листинге 6.4). Данные определялись как static, а следовательно, их нельзя увидеть извне. Функции в этом файле могли вызываться из других файлов и обращаться к данным от имени функций-клиентов.

В C++ данные и функции объединяются в классы, что уменьшает необходимость применения глобальных переменных. Пространства имен также снижают потребность в глобальных переменных. Следовательно, в C++ такая техника программирования не столь важна, как в языке C. Тем не менее при определении глобальных переменных не следует забывать о static, чтобы свести к минимуму взаимодействие с другими разработчиками и передать информацию о коммуникациях между функциями сопровождающему приложение программисту.

Второй смысл ключевого слова static несколько иной. При применении к локальной переменной, определенной в функции или блоке (не забывайте, что по умолчанию эти переменные автоматические), данное ключевое слово перемещает

переменную из стека в фиксированную область памяти. Там переменная будет существовать не от начала до конца функции или блока (как автоматические переменные), а от начала программы до конца ее выполнения, т. е. значение в данной области памяти присваивается при первом входе в область действия и будет доступно при повторном входе в нее. Что касается имени переменной, то к нему все равно применимы правила области действия, о которых рассказывалось выше. Имя неизвестно вне фигурных скобок — блока, где она определена. Следовательно, другие независимые области действия (даже в том же файле) могут использовать это имя для своих целей. Более того, несколько переменных в разных областях действия могут определяться как static и носить одно и то же имя. Это не приведет к конфликту имен, хотя память для переменных отводится в фиксированной области. Поскольку они находятся в разных областях действия, эти имена известны в разные моменты выполнения программы.

Например, функцию printAccounts() в листинге 6.3 можно модифицировать для печати только одного счета. Для этого удобно определить глобальную переменную i и использовать ее как индекс в printAccounts():

```
const int MAX = 5;
Account a[MAX];                          // глобальные данные для обработки
int count = 0;                           // число элементов в наборе данных
int i;                                   // глобальный индекс
...
void printAccounts()

{ cout << a[i].num << " " << a[i].bal << endl;
  i++; }                                 // увеличить индекс после использования
```

В функции main() удобно вызывать функцию printAccounts() в цикле:

```
for (int j = 0; j < count; j++)
  printAccounts();
```

Язык позволяет использовать индекс i и в цикле, но текущая версия моего компилятора не дает возможности делать это. (В последнем цикле в листинге 6.3 определяется i.) Недостаток такого подхода состоит в использовании большего числа глобальных переменных ("загрязнении" глобального пространства). Чтобы избежать этого, можно переместить определение индекса i в printAccounts() и избежать потенциальных конфликтов с другими случаями использования этого имени:

```
void printAccounts()
{ int i = 0;
  cout << a[i].num << " " << a[i].bal << endl;
  i++; }                                 // увеличение индекса после использования
```

Теперь индекс является автоматической переменной. Он получает новое пространство в стеке при каждом вызове функции printAccounts() из main(). Следовательно, значение индекса из предыдущего вызова не запоминается. Кроме того, индекс устанавливается в 0 при каждом вызове данной функции. Ключевое слово static решает обе проблемы:

```
void printAccounts()
{ static int i = 0;
  cout << a[i].num << " " << a[i].bal << endl;
  i++; }                                 // увеличение индекса после использования
```

На первый взгляд, в этом нет смысла. Как увеличивать индекс, если значение i сбрасывается в 0 при каждом вызове? Но это не так, как вы подумали.

В предыдущей версии printAccounts() начальное значение присваивалось i при каждом вызове. В данной версии, поскольку i — статическая переменная, она присваивается только один раз, хотя, казалось бы, это делается при каждом вызове цикла. На самом деле присваивание происходит не при первом вызове функции printAccounts(), а когда распределяется память для всех глобальных переменных — перед первым оператором в функции main(). Когда вызывается функция printAccounts(), оператор инициализации пропускается и в следующем операторе используется предыдущее значение данной локальной переменной:

```
void printAccounts()
{ static int i = 0;                          // выполняется только один раз
  cout << a[i].num << " " << a[i].bal << endl;
  i++; }                                     // выполняется при каждом вызове
```

В этом случае в явной инициализации даже нет необходимости. Переменные static неявно инициализируются нулем, и следующая версия функции printAccounts() вполне законна:

```
void printAccounts()
{ static int i;                    // неявная инициализация нулем
  cout << a[i].num << " " << a[i].bal << endl;
  i++; }                           // увеличение индекса после использования
```

Сопровождающему приложение программисту придется немного подумать, чтобы понять, почему эта функция обновляет переменную, которая явно не инициализировалась. Предыдущий вариант менее эффектен, но лучше передает намерения разработчика.

Применение локальных переменных static не считается хорошей практикой программирования. Оно требует чрезмерной координации клиентских и серверных функций, затрудняет понимание программы. Кроме того, в нем редко возникает необходимость. В большинстве случаев нетрудно найти решение, не требующее применения локальных переменных static. Например, способ печати остатков на счетах в листинге 6.3 (и в предыдущих версиях программы) прост, и в локальных переменных static нет нужды.

Глобальные функции static аналогичны статическим глобальным переменным в том смысле, что они не могут вызываться вне того файла, где определены — имя функции static невидимо для других файлов. Это означает, что данное имя можно использовать в другом файле для каких-то иных целей, без всяких конфликтов имен и прочих помех. Если функция вызывается только теми функциями, которые находятся в том же файле, где она определяется, то полезно явно определить ее как статическую, сделав видимой только в этом файле, а не во всей программе. Так, в листинге 6.3 функцию printAccounts() можно сделать статической:

```
static void printAccounts()
{ static int i = 0;
  cout << a[i].num << " " << a[i].bal << endl;
  i++; }                           // увеличение индекса после использования
```

Функцию printCaption() из листинга 6.4 также можно определить как static:

```
static void printCaption()              // вызывается только из этого файла
{ cout << caption; }
```

Подобно статическим глобальным переменным, здесь возникает проблема конфликта имен и передачи идеи разработчика сопровождающему приложение программисту. Поместив несколько функций-серверов в один файл с вызывающими их клиентскими функциями и определив их как статические глобальные функции, можно позволить программистам работать с другими файлами и использовать

эти имена функций без координации. Кроме того, это явно говорит сопровождающему приложение программисту, что в других файлах нет функций, зависящих от данного файла. Размещение клиентских и серверных функций в одном файле не всегда возможно и желательно. Когда это делается, следует документировать данные действия, определяя функции-серверы как static.

В использовании класса памяти static для связанных с классами функций есть еще один аспект. Они могут обращаться только к статическим полям класса. Подробнее о полях и функциях static рассказывается ниже.

Управление памятью: использование динамически распределяемой области

Правила области действия и разнообразные классы памяти в C++ прошли долгий путь эволюции, чтобы помогать программистам управлять памятью, выделяемой для программных объектов. Однако эти средства не решают проблему адекватной реализации динамических структур данных.

Реализация массивов динамических структур данных с контрольным значением или счетчиком допустимых записей — мощный и простой инструмент. Когда число элементов в наборе данных увеличивается или уменьшается, они позволяют добавлять или удалять компоненты. Однако максимальный размер данных должен быть известен на этапе компиляции. Выбор неоптимального размера влечет опасность переполнения или напрасной траты памяти.

Динамическое управление памятью решает эту проблему, поскольку память выделяется и освобождается динамически. Когда набор данных заполняет все пространство в массиве, динамически выделяется массив большего размера, туда копируются данные, а старый массив освобождается. Если данных становится меньше и слишком много памяти теряется впустую, выделяется массив меньшего размера, данные копируются в этот новый массив, а массив большего размера освобождается. Такой метод устраняет опасность переполнения и непроизводительной траты памяти.

Еще одна проблема непрерывных массивов состоит в том, что они эффективны только тогда, когда новые элементы добавляются в конец массива. Если нужно включить новый элемент в начало или в середину массива, придется сдвигать остальные элементы в конце массива.

Когда же элемент удаляется из середины массива, нужно также сдвигать остальные элементы к началу массива, убирая "пробел". Другой подход состоит не в устранении пробела в середине массива, а в использовании еще одного контрольного значения, обозначающего удаленные элементы. Это позволяет избежать необходимости сдвига при удалении, но требует дополнительной проверки каждого элемента при поиске. Если массив короткий, то вставка или удаление из середины массива случаются нечасто и эти недостатки несущественны. Для длинных массивов частая вставка и удаление элементов может отрицательно повлиять на производительность.

Одно из возможных решений данных проблем состоит в отказе от массива как механизма выделения памяти сразу для большого числа элементов. Вместо этого можно выделять память для элемента, только когда он вставляется в набор данных. Отдельные элементы связываются указателями, содержащими адреса динамически распределяемых элементов. С помощью операций с указателями можно вставлять элемент в набор данных, не тратя время на сдвиг других элементов. Когда элемент нужно удалить, его память освобождается для других целей. Операции с указателями также позволяют устранить пробел без сдвига элементов и пометки элемента как удаленного.

Применение указателей для динамических массивов и связанных структур данных — полезный и популярный метод. Но он сложнее, чем использование массивов фиксированного размера, о которых рассказывалось выше. Ошибки в работе с указателями — это ошибки этапа выполнения, нередко их трудно обнаружить. Частое выделение и освобождение памяти может отрицательно повлиять на производительность.

Во многих языках (в частности, в Lisp, Eiffel и Java) управление памятью считается слишком важным для целостности программ, чтобы можно было доверять его делающему ошибки программисту. В этих языках применяется так называемая автоматическая "сборка мусора": оценивается использование программой памяти и освобождаются области, которые нужно вернуть в пул памяти. На самом деле эти механизмы довольно медленные, сложные и неточные.

В C++ по аналогичной причине используется противоположный подход. Управление памятью здесь считается слишком важным для производительности программы, чтобы его можно было доверять общим (и часто неэффективным) алгоритмам. В C++ программист может полностью управлять распределением и освобождением памяти. Если программист делает ошибки, они могут привести к порче или "утечкам" памяти и аварийному завершению программы. Это плохо, но хорошие программисты не делают много ошибок. Алгоритмы динамического распределения памяти просты. При аккуратной реализации они вполне надежны. Кроме того, стандартные библиотеки предусматривают соответствующие структуры данных и функции, помогающие программисту избежать ошибок.

Сегмент памяти программ для хранения динамических данных, выделяемый в результате явного запроса, называется *динамически распределяемой памятью* (heap — куча). Именно нечто похожее на "кучу" представляет собой память после многократного выделения и освобождения. Структура динамически распределяемой памяти упрощает поиск свободной памяти подходящего размера для удовлетворения следующего запроса.

Размер фиксированного сегмента данных для глобальных и статических переменных вычисляется в процессе компиляции и компоновки. Размер стека и динамически распределяемой области не может вычисляться автоматически. Обычно стек и динамически распределяемая область памяти увеличиваются во встречном направлении (чтобы избежать преждевременного переполнения).

Между переменными, выделяемыми в динамически распределяемой области, и обычными переменными есть два различия, о которых уже упоминалось выше.

- Память для обычных переменных (распределяемых в стеке и фиксированной области памяти) выделяется согласно правилам языка. Память для переменных динамически распределяемой области выделяется с помощью явных операций, указываемых программистом.
- Обычные переменные имеют имена — псевдонимы (мнемонические ссылки) своих ячеек памяти. Переменные, распределяемые в динамической области, имен не имеют. На них можно ссылаться посредством указателей.

Указатели C++ как типизированные переменные

Указатель — это переменная, содержащая адрес другой переменной, которая может быть обычной (именованной) стековой переменной. Указатели, как правило, ссылаются на переменные, распределяемые в динамической области (неименованные). Сами указатели обычно размещаются в фиксированной секции данных (как глобальные или статические переменные) или в стеке (автоматические). Использование для указателя динамически распределяемой области — очень редкое явление. Указатели представляют собой просто именованные переменные.

В C++ указатели в основном используются:

- Для динамического распределения массивов, размер которых задается на этапе выполнения
- Для создания динамических структур данных, состоящих из несмежных связанных узлов
- Для передачи параметров функциям

В главе 1 уже рассказывалось об общем синтаксисе, семантике указателей и приводились некоторые примеры их использования (соответствующие первым двум из трех перечисленных пунктов). Передача параметров функциям обсуждается ниже.

Для создания переменной-указателя, во-первых, задается, что это указатель, во-вторых, указывается тип переменной, на которую он ссылается. Указатели C++ могут ссылаться (с помощью косвенной ссылки) на переменную только одного типа. Тип задается при определении указателя-переменной. Изучив более сложные аспекты C++, такие, как наследование и полиморфизм (хотя пока эти слова мало что говорят), можно понять, что данное правило имеет некоторые интересные исключения, но сейчас полезно запомнить, что указатель определяется как указатель на целое и должен ссылаться на значение типа `int`, а не `double`. Соответственно указатель на переменную `double` должен указывать на значение `double`, а не на целое.

Для обозначения переменной-указателя используется звездочка (*). Она следует за именем типа или перед именем переменной; пробелы между звездочкой и именем переменной или типа не имеют значения:

```
int * pi; char* pc; double*pd;          // годится любое число пробелов
```

Обратите внимание, что звездочка здесь — даже не операция, а просто обозначение того, что переменная — это указатель на тип слева от звездочки. Объявления с указателями следует читать справа налево. Например, `pi` — указатель на переменную типа `int`, а `pc` — указатель на переменную типа `char`. Можно сказать также, что `pi` имеет тип `int*` или `pc` — переменная типа `char*`. Как будет показано позднее, последний способ достаточно полезен.

Но в данных выражениях звездочка — не просто обозначение, это операция, применяемая к переменной-указателю (`pi`, `pc` и т. д.) для получения значения базового типа. Это значение по адресу, на который ссылается переменная-указатель. Имя операции-звездочки, считывающей значение по указателю, называется операцией разыменования (dereference) или снятия косвенности.

Если получение адреса значения называется указанием, то получение значения из адреса — это "разуказание", а не разыменование, но в C++ терминология заимствована из языка C, а C разрабатывался как язык высокого уровня для программистов, использовавших ассемблер. В ассемблере программисты называли вещи так, как им нравится, так что простым смертным их не понять.

Областью действия указателя-звездочки будет просто одна переменная-указатель. Она применяется к следующему за звездочкой идентификатору, а не к предшествующему имени типа. Это отличается от других определений и объявлений. Например, ниже только `pc` является указателем на `char`: `pchar` имеет тип `char`, а не `char*`.

```
char* pc, pchar;                // pchar имеет тип char, а не char*
```

Это очень распространенная ошибка. Чтобы сделать `pchar` указателем, нужно записать:

```
char* pc, *pchar;               // both pc и char - указатели
```

Указатели — обычные именованные переменные, автоматические или глобальные. Для них выделяется достаточно памяти, чтобы вместить адрес конкретного типа. Часто размер указателя совпадает с размером целого, однако рассчитывать на это не стоит. Операция `sizeof` может точно сообщить, так ли это на конкретной машине, но было бы неразумно писать программу, полагающуюся на размер указателя. Она не будет переносимой.



Осторожно! Переменные-указатели (адреса) часто имеют размер целого, независимо от типа значений, на которые они указывают. Не используйте размер указателя в своей программе — это может сделать программу непереносимой.

Подобно другим переменным, указатели по определению не имеют полезного значения (содержат 0, если они глобальные, и не определены, если автоматические). Указатели могут содержать адреса следующих объектов:

- Встроенных типов (`char* pc`)
- Типов, определяемых программистом (`Account* pa`)
- Массивов встроенного или определяемого программистом типа (обозначение то же, что и для переменных, например `char* pc` или `Account* pa`);
- Функций (пока еще рано описывать указатели-функции)
- Другие указатели; так `char** pcc` можно использовать как указатель на символьный указатель (как `pc` выше). Здесь лишь упоминается, что такое возможно, но не торопитесь применять это в своих программах

Для доступа к значению объекта, на который ссылается указатель, нужно произвести операцию разыменования (звездочку) к указателю как префиксную операцию (слева от имени указателя). Другими словами, указатель разыменовывается. Например, ниже в переменную типа `double` (на нее ссылается указатель `pd`) помещается значение 5.0, значение 20 помещается в переменную типа `int`, на которую указывает `pi`, а значение `'a'` — в символьную переменную с указателем `pc` (если значение типа `double` по указателю `pd` положительно, а это так):

```
pd = 5.0; *pi = 20; if (*pd > 0) *pc = 'a';          // нехорошо
```

Как уже упоминалось выше, если `pi` — указатель на `int`, то `*pi` имеет тип `int`. Аналогично `*pd` имеет тип `double`. С точки зрения типов значений пример корректен, однако эти указатели нигде не инициализируются, а разыменование неинициализированных указателей недопустимо.

Если не инициализируется глобальный указатель, он содержит 0. Разыменование нулевого указателя немедленно завершает программу:

```
pd = NULL; *pd = 5.0;          // нулевой указатель - исключение
```

Если локальный указатель (автоматическая переменная) не инициализирован, он содержит случайный набор битов (как и любая другая автоматическая переменная). Этот набор можно интерпретировать как адрес, но такой адрес будет указывать на произвольное место в памяти. Считывание значения по данному адресу даст просто "мусор", а запись по нему приведет к порче содержимого памяти, может вызывать сбой операционной системы, исключительную ситуацию при выполнении программы, дать некорректные результаты или даже корректные результаты (если такая ячейка памяти программой не используется). Применение неинициализированных указателей — распространенная ошибка, которую очень трудно диагностировать, поскольку они могут указывать на любую область памяти.

Неинициализированные указатели могут ссылаться на произвольную область памяти, а это способно привести к порче памяти или некорректным результатам. В C++ подобные ошибки неинициализированных указателей являются ошибками этапа выполнения, а не этапа компиляции. Это неприятно: если возникает данная ошибка, дружелюбный компилятор не говорит, что ее нужно исправить. Приходится догадываться об ошибке, анализируя результаты тестового выполнения программы.

Указатели могут инициализироваться с помощью операции адреса (&). Некоторые примеры операций с указателями приведены в листинге 6.5. Здесь в функции `main()` определяются две автоматические переменные — типа `int` и `char`. Кроме того, определяются два указателя на `int` и `char`, инициализируется указатель символьного типа для ссылки на символьную переменную и присваивается указатель целого типа для ссылки на целое. После этого с помощью разыменования указателя целому присваивается новое значение. Потом программа проверяет значение целого, используя разыменованный указатель, и присваивает с помощью разыменованного символьного указателя символьное значение. В конце программа устанавливает символьный указатель на целочисленное значение.

```
i как десятичное: 28791
i как шестнадцатеричное: 7077
i через указатель int:  28791
i через указатель char: 119
i через указатель char в hex: 77
```

Рис. 6.4. *Вывод программы из листинга 6.5 (обратите внимание на некорректный доступ к int)*

Большинство компиляторов запрещают прямое присваивание вида `pc = &i`. Действительно, `pc` имеет тип `char*`, а `&i` — тип `int*`. C++ здесь строг: он допускает неявное преобразование между числовыми типами, но не между указателями разных типов. Чтобы присваивание указателей было законным, они должны быть одного типа. Между тем, можно без ограничений использовать явное преобразование указателей разных типов (приведение типа). Предполагается, программист знает, что он делает. Приведение типов указателей — опасная практика. Путем разыменования символьного указателя можно обращаться к частям целого и изменять их. Из рис. 6.4 видно, что разыменование двух указателей на одну целочисленную переменную дает разные результаты (в зависимости от типа указателя).

Листинг 6.5. Использование указателей с обычными именованными переменными

```
#include <iostream>
using namespace std;

int main()
{
  int i; int pi; char *pc;                    // неинициализированные указатели
  pi = &i;                                    // указатель на i
  *pi = 502;                                  // нормально, i = 502
  if (*pi>0) *pc = 28791;                     // то же, что и if(i>0) i=28791
  pc = (char*) &i;                            // в некоторых компиляторах не требуется
  int a1 = *pi;                               // доступ к i через указатель
  int a2 = *pc;                               // доступ к i через указатель
  cout   << " i как десятичное: " << i << endl
         << " i как шестнадцатеричное: " << hex << i << endl;
  cout   << " i через указатель int: " << dec << a1 << endl;
  cout   << " i через указатель char: " << a2 << endl;
  cout   << " i через указатель char в hex: " <<hex << a2 << endl;
  return 0;
}
```

В листинге 6.5 hex и dec называются *манипуляторами*. Они указывают объекту cout основание системы счисления для вывода значений (шестнадцатеричное или десятичное). Как и манипулятор endl, они вставляются в поток вывода и изменяют его характеристики. Как видно из рис. 6.4, значение, считываемое указателем pi, корректно (28791), а значение, получаемое символьным указателем pc,— нет. Как показывает вывод в шестнадцатеричном виде, символьный указатель считывает только часть битовой последовательности (77 в шестнадцатеричном виде) значения i (7077 в шестнадцатеричном представлении). По существу, указатель целого типа может видеть все значение, а символьный указатель — только один байт. Ни один из них не извлечет правильно значение типа double. Вот почему важно разыменовывать указатели корректных типов.



Советуем При разыменовании указателя убедитесь, что его тип соответствует типу значения, на которое он ссылается. В противном случае получаемое с помощью указателя значение будет некорректным.

Операции с указателями нельзя назвать интуитивно понятными. Очень трудно следить за операциями с указателями, читая программу, поэтому важно помочь своей интуиции, рисуя картинки. Здесь будут полезны два вида рисунков: один, показывающий переменные, для которых память выделяется в стеке (рис. 6.5a), а другой — демонстрирующий, какие указатели на какие значения указывают (рис. 6.5b).

Рис. 6.5a демонстрирует целое i, целочисленный указатель pi и символьный указатель pc, для которых память распределена в стеке. Хотя их реальный размер может быть одним и тем же, принято представлять указатели маленькими прямоугольниками. Здесь показано значение, содержащееся в целом i. Указатели pi и pc содержат адрес i, но этот адрес нам не важен. Вместо адреса используются стрелки, показывающие, что указатели ссылаются на один адрес. Хотя стрелки показывают на разные места, это допустимое приближение. Неизвестно, содержит указатель адрес старшего или младшего байта. Отмечено лишь, что указатели ссылаются на значение, а разыменование этих указателей позволяет получить значение (если тип указателя корректен).

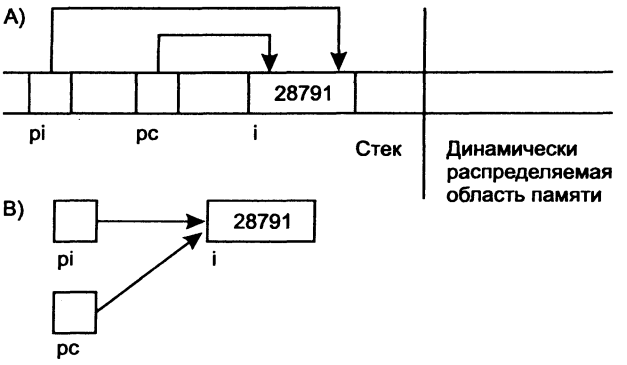


Рис. 6.5. *Указатель целого типа и символьный указатель, ссылающийся на именованную целочисленную переменную i, которая распределяется в стеке*

На рис. 6.5b показана та же конфигурация без указания, где распределяются переменные i, pi и pc — в стеке или в динамически распределяемой памяти. Рабочее предположение должно быть таким: если задаются имена переменных, они распределяются в стеке. Если имена не задаются, они распределяются в динамической области памяти. Стрелки показывают, что указатели содержат адреса переменных. Следовательно, их можно использовать для доступа к данным переменных.

Из рисунка видно, что применение указателей для операций с именованными переменными не особенно полезно. Использование указателей для такого вида операций с данными не более эффективно, чем обычная работа с переменными (в данном примере i), на которые ссылаются эти указатели. Установки указателей на значения несоответствующих типов приводит к излишней сложности и ошибкам. Некоторые программисты используют аналогичные методы в вызовах функций, чтобы избежать операции получения адреса (это будет продемонстрировано в следующей главе). Но пока будем считать, что указатели нужны не для этого. Они нужны для распределения памяти в динамической области.

Выделение памяти в динамической области

Операции C++ в основном представляют собой простые символы, но в данном языке больше операций, чем специальных символов на клавиатуре, а потому в C++ применяются двух- и даже трехсимвольные операции. Но и этого оказалось недостаточно — в C++ для некоторых операций зарезервированы слова new и delete. Они обозначают унарные операции, т. е. имеют только один операнд. Эти операции используются для управления памятью в динамически распределяемой области. Динамически распределяемая область памяти — тоже просто терминология. Программист не знает, где она находится. Что такое динамически распределяемая область? Это область памяти, где операции new и delete выделяют и освобождают память. Нужно лишь знать, что выделенная память в подходящее время должна освобождаться.

В операции new в качестве операнда используется имя типа. Она запрашивает у ОС выделения объема памяти, вмещающего значение типа, который задается в операнде. Если память выделена успешно, операция new возвращает адрес памяти, выделенной операционной системой в динамически распределяемой области. Значение адреса обычно присваивается указателю соответствующего типа, и этот указатель может использоваться для операций с неименованной выделенной памятью. Если ОС не хватает памяти, то new возвращает вместо адреса нулевое значение. Программа может проверить результат (возвращаемое значение) и решить, что делать дальше (например, вывести сообщение и завершить работу).

В операции delete операндом является имя указателя. Она находит в динамически распределяемой области выделенную память (размер которой соответствует типу указателя) и просит ОС пометить ее как неиспользуемую. Очень важно, чтобы каждой операции new соответствовала операция delete. Чтобы избежать утечек памяти, программа всегда должна освобождать выделяемую память.

Листинг 6.6 Использование указателей с неименованными переменными в динамически распределяемой области

```
#include <iostream>
using namespace std;

int main()
{
  int *pi; char* pc;                          // неинициализированные указатели
  pi = new int;                               // получение неименованной памяти и ссылка на нее
  if (pi == NULL)                             // в случае неудачи возвращает ноль
    { cout << "Нет памяти\n"; return 0; }     // или пытается восстановить
  pc = new char;                              // получение неименованной памяти и ссылка на нее
  if (pc == 0)                                // необходимая предосторожность
    { cout << " Нет памяти \n"; return 0;}    // или попытка восстановления
  *pi = 28791;
  if (*pi > 0) *pc = 'a';                     // операции с неименованными объектами
  cout << " целое в динамической области: " << * pi << endl;
  cout << " символьное значение в динамической области: " << *pc << endl;
  delete pi; delete pc;                       // часть жизненного цикла
                                              // динамически распределяемой области
  cout << " (после удаления) int: " << pi << "char: << "*pc << endl;
  return 0;
}
```

В листинге 6.6 приведены примеры использования данных операций. В функции main() определяются два указателя: pi типа int и pc типа char. Затем они инициализируются с помощью операции new. Программа проверяет, успешно ли

```
целое в динамической области: 28791
символьное значение в динамической области: a
(после удаления) int: -572662307  char: ||
```

Рис. 6.6. *Вывод программы из листинга 6.6*

выделена память. Если нет, программа завершает работу, так как ей не удается достичь своих целей. Нередко в этом случае предпринимаются некоторые меры для корректного завершения (сохранение данных). Иногда программа может попытаться освободить какую-то память, чтобы продолжить работу в специальном режиме с ограниченной функциональностью. Как видно из рис. 6.6, распределение памяти прошло успешно, а через указатели корректно присваиваются и считываются значения (целое 28791 и символьное 'a').

В данном примере NULL — библиотечная константа. Многие программисты предпочитают использовать эту константу, а не числовое значение 0, указывая тем самым, что программа работает с указателями. Другие применяют числовой 0. Результат будет один и тот же. Важно помнить, что за операцией new должна следовать проверка на успешное выделение памяти.

Операция delete возвращает память, выделенную операцией new, в динамически распределяемую область. Она достаточно "интеллектуальна" и знает тип своего указателя-операнда, а потому освобождает в точности столько памяти, сколько было выделено по new. Если программист забывает о delete, программа все равно будет работать, но со временем может исчерпать всю память в динамически распределяемой области и при очередном использовании new будет возвращаться 0 (особенно если приложение работает продолжительное время). Программисту очень важно не забывать освобождать всю память, запрашиваемую программой из динамически распределяемой области.

Прочитайте программу, содержащую операции удаления, громко вслух. Скажите "delete pi, delete pc". Замечательно. Но не стоит убеждать себя, что указатели при этом действительно удаляются. Удаляется (освобождается) только неименованная область памяти соответствующего размера, на которую ссылаются pi и pc. Указатели здесь представляют собой именованные стековые переменные, а память для них распределяется в соответствии с правилами области действия, о которых рассказывалось ранее. Память выделяется при определении (здесь в начале функции main()) и освобождается при завершении области действия, т. е. когда выполнение достигает завершающей фигурной скобки области действия (здесь закрывающей фигурной скобки функции main()).

Удаляются только неименованные переменные динамически распределяемой области. Не стоит пытаться удалять именованные переменные, для которых память выделяется в стеке, например переменную i в листинге 6.5 (через указатель pi, указатель pc или непосредственно без указателей).

После удаления переменной динамически распределяемой области, на которую ссылается указатель, этот указатель снова становится неинициализированным и не должен использоваться для разыменования. В конце листинга 6.6 делается попытка извлечь значения по указателям pi и pc. Как показано на рис. 6.6, эти указатели теперь ссылаются на произвольные значения, а не на те, куда они должны указывать. Стоит отметить, что компилятор на это никак не реагирует и не сообщает об ошибке. Операционная система также не препятствует подобным действиям (хотя могла бы и даже должна).

И еще пара слов о delete. Не следует использовать эту операцию с неинициализированным указателем. Нужно применять только указатель, ссылающийся на память в динамически распределяемой области, выделенную операцией new. Например, двукратное освобождение памяти даст ошибку этапа выполнения (а не компиляции):

```
delete pi; delete pi;                    // некорректно
```

Этот код некорректен в том смысле, что его поведение неопределенно. Он может привести к аварийному завершению программы, дать неправильные результаты или даже вполне правильные — все, что угодно. Нужно тщательно следить за

тем, что вы делаете с управлением памятью. Особенно это касается циклов. Не удаляйте дважды один и тот же указатель. Удаление указателя NULL допускается и не дает никакого эффекта.

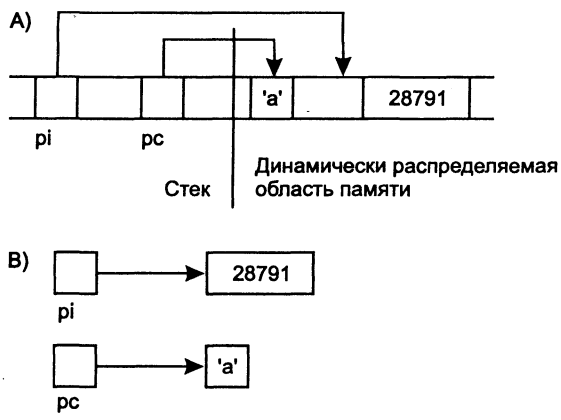


Рис. 6.7. *Указатель целого типа и символьный указатель, ссылающиеся на неименованную целочисленную переменную и символьную переменную в динамически распределяемой области*

На рис. 6.7 показана работа с памятью согласно с листингом 6.6. На рис. 6.7а, 6.7б демонстрируются указатели pc и pi, распределяемые в стеке, неименованное целочисленное значение и символьное значение в динамической области. Указатели являются именованными (память для них выделяется в стеке), а целочисленная и символьная переменная — не именованными. Можно приблизительно представить, что целочисленная и символьная переменная имеют разные размеры, но об указателях такдумать не стоит. Кажется, что указатели обычно меньше, чем значения, на которые они ссылаются (даже когда для них выделяется больше памяти).

Операции new и delete доступны только в C++, но не в C. В языке C для динамического распределения памяти используется вызов библиотечной функции malloc(), а возвращается память при вызове функции free(). Функция malloc() менее интеллектуальна, чем операция new. Она ничего не знает о размерах типов данных, а поэтому в аргументе нужно указывать, сколько байтов освобождается. Кроме того, она возвращает обобщенный указатель, так называемый указатель void, который нельзя разыменовывать. Возвращаемый функцией malloc() указатель нужно преобразовывать в соответствующий тип с помощью операции приведения типа. Если распределение памяти выполнить не удается, malloc() возвращает указатель NULL. Таким образом, программа может проверить, действительно ли доступна запрошенная память. Так как C++ обратно совместим с языком C, функция malloc() поддерживается и здесь. Она определена в стандартной библиотеке cstdlib (или stdlib.h).

```
pi = (int*) malloc(sizeof(int));        // получить неименованную память в
                                        // динамически распределяемой области
```

В качестве аргумента функции free() используется указатель. Она достаточно интеллектуальна и знает, сколько байтов памяти нужно освободить.

```
free(pi);                               // возвращает память динамически
                                        // распределяемой области для других целей
```

В листинге 6.7 показаны те же операции, что и в программе из листинга 6.5, но с использованием функций malloc() и free(). Обратите внимание на включение заголовочного файла cstdlib.h. Результат программы будет таким же, как на рис. 6.5.

Многие компиляторы C++ в действительности реализуют операции new и delete в терминах функций malloc() и free(), однако в C++ new и delete используются намного чаще, чем malloc() и free(). Они проще, и, кроме того, эти функции применяются для управления памятью при работе с такими объектами, как классы, где вызываются неявно через конструкторы и деструкторы. Этого нельзя сказать о функциях malloc() и free(). Но есть и общий момент. И операции, и библиотечные функции malloc() и free() используются парами. Если память выделяется с помощью функции new, то ее нельзя освободить функцией free(), а если функция выделялась функцией malloc(), ее нельзя освободить функцией delete. Компилятор не обнаруживает ошибок такого типа. Чтобы избежать подобных ошибок, многие программисты применяют только операции new и delete и избегают использования malloc() и free().

Листинг 6.7. Использование функций `malloc()` и `free()`, служащих для управления памятью

```
#include <iostream>
#include <cstdlib>                          // заголовочный файл для malloc() и free()
using namespace std;

int main()
{
  int *pi; char* pc;                        // неинициализированные указатели
  pi = (int*) malloc(sizeof(int));          // получение неименованной памяти
                                            // и ссылка на нее
  if (pi == NULL) // в случае неудачи возвращает ноль
    { cout << "Нет памяти\n"; return 0; }   // или попытается восстановить
  pc = (char*) malloc(sizeof(char));        // получение неименованной памяти
                                            // и ссылка на нее
    if (pc == NULL)                         // необходимая предосторожность
  { cout << "Нет памяти\n"; return 0; }     // или попытка восстановления
  *pi = 28791;
  if (*pi > 0) *pc = 'a';                   // операции с неименованными объектами
  cout << " целое в динамической области: " << *pi << endl;
  cout << " символьное значение в динамической области: " << *pc << endl;
  free (pi); free(pc);
  cout << " (после удаления) int: " <<*pi <<"char: << "pc << endl;
  return 0;
}
```

Между тем есть большое количество унаследованных программ C (и C++), где применяются функции `malloc()` и `free()`. Судя по долговечности программ, вызвавших проблему 2000 г., следует быть готовым к тому, что еще долгие годы придется иметь дело с этими функциями.

Как видно, использование динамической области для управления памятью при работе со значениями встроенных типов не очень полезно. Можно добиться того же результата с помощью именованных переменных в стеке.

Некоторые программисты освобождают память, занимаемую в динамической области отдельными значениями. Программа в этом случае компилируется и выполняется корректно, но она будет несколько сложнее, чем должна быть. Динамическое управление памятью для индивидуальных переменных вынуждает следить за правильным выбором момента выделения и освобождения памяти. Сложность программы еще более увеличивается — в дополнение к определению, инициализации и разыменованию указателей. При этом не достигается никаких особых преимуществ. Следует избегать данной практики. Используйте динамически распределяемую область памяти только для динамических массивов и динамических структур данных.

Массивы и указатели

Необходимость указывать длину массива во время компиляции в C++ направлено на эффективное использование памяти и повышение производительности на этапе выполнения. В то же время это создает проблемы переполнения массивов и непроизводительного расходования памяти, поскольку во многих приложениях размер наборов данных известен только на этапе выполнения. Между тем синтаксис C++ допускает в качестве размеров массивов только константы. Именно в таких ситуациях полезно динамическое распределение памяти.

Чтобы можно было использовать динамически распределяемые массивы, следует изучить взаимосвязь между массивами C++ и указателями. Это соотношение основано на еще одном уникальном средстве C++: имя массива (без квадратных скобок и других модификаторов) означает то же самое, что адрес начального элемента массива. Следовательно, имя массива может использоваться для инициализации указателя соответствующего типа (того же, что и элемент массива). Содержимое указателя становится адресом первого элемента массива. Разыменование указателя приводит к считыванию (или изменению) первого элемента массива. Это открывает возможность использования указателя как синонима имени массива в вызовах функции и с индексами массива.

В следующем примере выделяется память для двух коротких символьных массивов `buf[]` и `data[]`, определяются два символьных указателя `p` и `q`. Указатели инициализируются с помощью адреса начального элемента массива. Данный пример показывает, что это можно сделать явно через адрес (`p = &buf[0];`) или неявно (с помощью имени массива (`q = data;`):

```
char buf[6], data[6], *p, *q;       // массивы и указатели
int i;
p = &buf[0];                        // явный синтаксис для адреса первого элемента
q = data;                           // неявный синтаксис для адреса первого элемента
for (i=0; i < 6; i++)               // присваивание компонентов массива
  { p[i] = 'A'+i;                   // буквы в верхнем регистре "ABCDEF"
    q[i] = 'a'+i; }                 // буквы в нижнем регистре
```

Единственная разница между указателем и именем массива в том, что указатель можно переназначить, и он будет ссылаться на другую ячейку памяти (с помощью операции получения адреса `&` или присваивания указателя), а имя массива — это константа. Ей нельзя присвоить другой адрес. В следующем примере первая часть массива `data[]` (символы в нижнем регистре) копируется во вторую часть массива `buf[]`, после чего он будет содержать буквы "ABCabc", а не "ABCDEF".

```
p = &buf[3];                        // указывает на вторую половину массива
for (i=0; i < 3; i++)               // замена первых трех компонентов
  p[i] = q[i];                      // то же, что buf[i+3]=data[i];
```

В обоих примерах имена указателей те же, что в именах массивов. Везде, где используется `q[i]`, можно применять `data[i]`. Это хорошо, но не очень практично, так как не позволяет сделать что-нибудь новое. Однако работа с массивами через указатели открывает гораздо большие возможности.

Еще одно уникальное средство C++ состоит в том, что при арифметических операциях с указателями учитывается тип и размер элементов памяти, на которые ссылается указатель. Например, если `ptr` — указатель на значение `double` по адресу 2000, то `ptr + 1` указывает на значение `double` после адреса, на который указывает `ptr` (2008, а не 2001).

Это особенно удобно, когда указатель ссылается на элемент массива. Увеличение указателя на 1 — не то, что вы думаете. При этом не добавляется 1 к содержимому указателя-переменной (как в случае арифметических операций с числовыми типами данных). Такая операция приводит к тому, что указатель будет ссылаться на следующий элемент массива. Разыменование указателя дает значение следующего элемента. Увеличение указателя на 2 перемещает указатель на два элемента массива. В следующем примере первая часть массива `data[]` (те же символы "abc") копируется в первую часть массива `buf[]`, так что содержимое его превращается в "abcabc".

```
p = buf;                            // снова указывает на начало массива
for (i=0; i < 3; i++)               // заменить первую половину массива
  *(p+i) = *(q+i);                  // также эквивалентно buf[i]=data[i];
```

Обратите внимание, что операция разыменования имеет более высокий приоритет, чем арифметические операции, поэтому `*p + i` здесь использовать не следует. Это означает `p[0] + i`, а не `p[i]`.

Еще более эффектный код можно написать с помощью применяемой к указателям операции инкремента (или декремента). Во всех случаях добавление 1 к указателю фактически означает сложение размера типа с указателем (т. е. с адресом, хранимым в указателе). В результате указатель перемещается на следующий элемент массива. В листинге 6.8 сведены предыдущие примеры. В первом цикле устанавливается и содержимое массива `buf[]` (ABCDEF), где вместо `buf[i]` используется `p[i]`. Во втором цикле модифицируется вторая половина массива. В этом цикле `p[i]` означает не `buf[i]`, а `buf[i+3]`. Третий цикл выводит массив `buf[]` (ABCabc) с помощью обычных обозначений. Затем указатель `p` устанавливается снова на начало массива `buf[]`, а четвертый цикл заменяет первую половину `buf[]`. Результат выводится путем применения к указателю операции инкремента. Вывод программы представлен на рис. 6.8.

```
Начальный буфер:          ABCDEF
Замененная вторая половина:   ABCabc
Замененная первая половина:   abcabc
```

Рис. 6.8. *Вывод программы из листинга 6.8*

Листинг 6.8. Использование указателей для обработки массива

```
#include <iostream>
using namespace std;

int main()
{
  char buf[6], data[6], *p, *q;          // массивы и указатели
  int i;                                 // индекс массива
  p = &buf[0];                           // явный синтаксис для адреса
  g = data                               // неявный синтаксис для адреса
  cout << "Начальный буфер: ";
  for (i=0; i < 6; i++)                  // присваивание компонентов массива
    { p[i] = 'A'+i;                      // буквы в верхнем регистре
      cout << p[i];                      // выводит ABCDEF
      q[i] = 'a'+i; }                    // q и data - синонимы
  p = &buf[3];                           // указывает на вторую половину
  for (i=0; i < 3; i++)                  // заменить последние три компонента
    p[i] = q[i];                         // то же, что buf[i+3]=data[i];
  cout << endl << "Замененная вторая половина: ";
  for (i=0; i < 6; i++)
    cout << buf[i];                      // выводит ABCabc
  p = buf;                               // указывает на начало массива
  for (i=0; i < 3; i++)                  // замена первой половины массива
    *(p+i) = *(q+i);                     // то же, что buf[i]=data[i];
  cout << endl << "Замененная первая половина: ";
  while (p - buf < 6)                    // увеличенный указатель
    cout << *p++;                        // не следует злоупотреблять этим средством
  cout << endl;
  return 0;
}
```

Когда операции инкремента и разыменования используются в одном выражении (как `*p++`), приоритет у них одинаковый, но они вычисляются справа налево, а не слева направо, как большинство операций C++ (см. таблицу 3.1 в главе 3). Между тем постфиксная операция передает значение указателю до инкремента. Следовательно, выражение `*p++` имеет следующий смысл: сохранить старый

указатель, увеличить его для ссылки на следующий элемент массива и вернуть значение по адресу, на который ссылается старый указатель. Другими словами, если temp — указатель символьного типа, то *p++ эквивалентно следующему:

```
(temp = p, p++, *temp)
```

Указатели и имена массивов эквивалентны во всех отношениях, кроме одного: указатель можно увеличивать (инкремент) или переназначать, а имя массива — нет. Например, в конце листинга 6.8 было бы ошибкой снова выводить массив buf[] таким образом:

```
while (p - buf < 6)          // смещение в элементах массива
  cout << *buf++;            // синтаксическая ошибка
```

Для этой цели можно без проблем использовать другой указатель:

```
q = buf;
while (p - q !=0)          // смещение в элементах массива
  cout << *q++;            // не следует злоупотреблять данным средством
```

Обратите внимание, что указатель p используется здесь как контрольное значение. В конце листинга 6.8 указатель p был увеличен для ссылки на позицию после завершающего элемента массива buf[]. Вот почему приведенный выше цикл завершается, когда указатель q заканчивает проход по элементам массива buf[] и ссылается на элемент после последнего, т. е. на то же место, что и указатель p.

Конечно, важно понимать взаимосвязь между обозначением указателя и обозначением массива. Существует немало унаследованного кода C и C++, где используется это средство, однако применение указателей нельзя назвать интуитивно понятным. Оно может легко запутать неопытных программистов, поэтому лучше использовать увеличение индекса, а не указателей. В то же время для многих применение указателей вместо индексов считается признаком хорошей квалификации программиста, ведь арифметические операции с указателями выглядят весьма эффектно.

В старые добрые времена операции с указателями не только красиво выглядели, но и позволяли создавать более быстрые программы, однако при современных компиляторах это уже не так. Оба метода генерируют один и тот же код.

Динамические массивы

До сих рассказывалось о способах использования указателей на именованные переменные в стеке, на неименованные переменные в динамически распределяемой области и на именованные массивы в стеке. Эти указатели делают программу излишне сложной и при этом не дают заметных преимуществ. Даже применение указателей с именованными массивами (как в приведенных выше примерах) не особенно полезно. Использование именованных массивов проще, чем работа с указателями.

Теперь рассмотрим примеры, где указатели действительно дают реальную выгоду. Они способны помочь в динамическом распределении памяти и при необходимости обойти проблему спецификации размера массива на этапе компиляции. Это делается с помощью динамически распределяемых массивов.

Листинг 6.9 показывает упрощенную версию программы, представленной в листинге 5.11, где обрабатываются введенные с клавиатуры суммы транзакций.

Метод, который использовался в листинге 5.11 (символьное контрольное значение в конце вводимых данных), можно считать хорошим решением для интерактивного ввода. В листинге 6.9 применялось нулевое контрольное значение. Цикл ввода завершался по break, когда вводилась нулевая сумма. Кроме того, цикл завершался при превышении счетчика введенных пользователем данных размера массива data[].

Листинг 6.9. Считывание данных транзакций с защитой от переполнения массива

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int NUM = 3;                          // для отладки: должно быть больше
  double amount, total = 0, data[NUM];
  int count = 0;                              // инициализация текущих данных
do {                                          // выполнение до конца файла или до переполнения
массива
    cout << "Введите сумму (или 0 для завершения): ";
    cin >> amount;                            // получить из файла следующее значение double
    if (count==NUM || amount==0) break;       // переполнение или равенство контрольному значению
    total += amount;                          // обработка текущих действительных данных
    data[count++] = amount;                   // и получение следующей строки ввода
  } while (true);
  if (amount != 0)                            // прочитаны ли все данные?
    { cout << "Нет памяти: ввод был прекращен\n";
      cout << "Значение " << amount << "не сохранено" << endl; }
  cout   << "\nОбщая сумма " << count << "значений равна "
         << total << endl;
  if (count == 0) return 0;                   // нет результатов, если нет ввода
  cout << "\nНом. транз. Сумма\n\n";
  cout.setf(ios::fixed);                      // установить формат fixed для double
  cout.precision(2);                          // всего цифр, если НЕ ios::fixed
  for (int i = 0; i < count; i++)             // снова проход по данным
    { cout << setw(4); cout << i+1)           // номер транз.
      cout << setw(11); cout << data[i] << endl; }        // знач. тразакции
  return 0;
}
```

Таким образом, цикл чтения может прерываться по двум причинам: конец ввода и переполнение массива. Если поведение программы должно прерываться для разных случаев завершения цикла, то нужно проверять, по какой причине завершен цикл. В данном примере программа выводит пользователю предупреждающее сообщение о переполнении массива. Проверку `count == NUM` здесь нельзя считать надежной, так как вводимый набор данных может содержать в точности столько записей, сколько элементов в массиве данных. Для реальных программ, обрабатывающих сотни и тысячи записей, такое не должно случаться часто, однако распространенная ошибка программистов — не проверять то, что случается очень редко. Рано или поздно происходят даже маловероятные события. Если же не проверять такие редкие события, могут быть проблемы.

```
Введите сумму (или 0 для завершения): 22
Введите сумму (или 0 для завершения): 33
Введите сумму (или 0 для завершения): 44
Введите сумму (или 0 для завершения): 55
Не хватает памяти: вывод прекращен
Значение 55 не сохранено

Общая сумма 3 значений равна 99

Ном. транз.   Сумма

   1          22.00
   2          33.00
   3          44.00
```

Рис. 6.9. *Вывод программы из листинга 6.9 (усечен)*

В листинге 6.9 проверяется наличие контрольного значения (сумма транзакции равна 0). При вводе такого значения цикл завершается. Если его не было, то можно предположить, что причиной завершения цикла стало переполнение массива. Если контрольное значение найдено, предполагается, что все данные считаны.

Вывод этой программы содержит только три записи массива. Они показаны на рис. 6.9.

Форматированный вывод здесь отличается от вывода программы из листинга 5.11. Функция setf() устанавливает управляющие флаги объекта cout. Она использует в качестве аргумента флаг ios::fixed, задающий вывод десятичных значений с фиксированной точкой, а не в экспоненциальном представлении (в виде мантиссы и порядка). У функции precision() в аргументе указывается число выводимых цифр при отображении значения на экране. Если флаг ios::fixed установлен, это число обозначает число цифр после десятичной точки (запятой), а если нет — общее число значащих цифр. Не путайте это.



Осторожно! Смысл аргумента функции precision() зависит от флага ios::fixed. Когда флаг установлен, аргумент означает число цифр после десятичной точки; в противном случае — это общее число значащих цифр.

В примере из листинга 5.11 функция width() задает минимальное число позиций, занимаемых на экране следующим выводимым значением. Если значению требуется для отображения больше позиций, то используются дополнительные позиции (тем самым форматирование не соблюдается). Другие функции форматирования, setf() и precision(), влияют на формат до следующего вызова данных функций с другим аргументом. Функция width() действует только на одно выводимое значение. После вывода значения устанавливается ширина по умолчанию (0), т. е. форматирование данных отсутствует. Таким образом, функция width() должна вызываться перед выводом каждого значения.

В листинге 6.9 используется манипулятор setw(), включаемый в поток вывода аналогично манипуляторам endl, dec и hex, о которых говорилось выше. Аналогично функции width(), областью действия этих манипуляторов является только одно значение вывода, поэтому манипулятор setw() нужно включать перед каждым значением, даже если ширина поля остается той же. Обратите внимание, что программа из листинга 6.9 использует заголовочный файл iomanip. В более ранних примерах также применялись манипуляторы (по крайней мере, endl), но для них этот заголовочный файл был не нужен. Он требуется только для манипуляторов, использующих аргументы. Если программист забудет включить его, программа компилироваться не будет. Увы, манипуляторы и форматтеры не согласованы, а потому программисту придется побеспокоиться о заголовочных файлах.



Осторожно! При использовании функций форматирования (например, width()) или манипуляторов без аргументов (таких, как endl) нет необходимости включать заголовочный файл iomanip. Если манипуляторы используются с аргументами (например, setw()), заголовочный файл iomanip нужно включить.

В листинге 6.10 показана программа, реализующая те же функции, что и программа из листинга 6.9, но программа в листинге 6.10 делает это с помощью динамически распределяемого массива. Вот где действительно указатели проявляют себя во всей красе: целостность программы сочетается с эффективностью ее выполнения.

Вместо выделения памяти для массива data[] в стеке путем указания заранее определенной константы этапа компиляции (например, 3), программа распределяет массив в динамической области с помощью задающей размер массива переменной size. Эта переменная имеет значение на этапе выполнения, а не на этапе компиляции. Учтите, что у динамического массива нет имени и к нему можно обращаться только через указатель data. Когда указатели используются для ссылки на именованные массивы (см. листинг 6.8), приходится выбирать между

именем указателя (например, p[i]) или именем массива (например, buf[i]). Здесь выбора нет. Массив в динамически распределяемой области памяти имени не имеет, поэтому нужно использовать для доступа к его элементам имя указателя.

Листинг 6.10. Считывание данных в массив, распределяемый в динамической памяти

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  const int NUM = 3;                          // для отладки: должно быть больше
  double amount, total = 0, *data;
  int count = 0, size = NUM;
  data = new double[size];                    // инициализация массива в heap
do {
  cout << "Введите сумму (или 0 для завершения): ";
cin >> amount;                                // получить из файла следующее значение double
  if (amount==0) break;                       // остановка по контрольному значению
  if (count == size)                          // нет памяти, запросить больше
    { size = size * 2;                        // сделать заметным
      double *q = new double[size];           // удвоенный размер массива
      if (q == 0)
         {  cout << "Нет памяти: ввод был прекращен << endl;
            break; }
      else {
         cout << "Выделено больше памяти: размер = " << size << endl;
      for (int i=0; i < count; i++)           // копирование старых данных
         q[i] = data[i];                      // запись с индексами
      delete [] data;                         // не забывайте об освобождении старых данных
      data = q; }                             // подключить основной указатель
  total += amount;                            // обработать текущие действительные данные
  data[count++] = amount;                     // и получить следующее значение
    } while (true);
  if (amount != 0)
    { cout << Нет памяти: ввод был прекращен\n";
      cout << "Значение " << amount << "не сохранено" << endl; }
  cout << "\nОбщая сумма " << count << "значений равна " << total << endl;
  if (count == 0) return 0                    // нет результатов, если нет ввода
  cout << "\nНом. транз. Сумма\n\n";
  cout.setf(ios::fiexed);                     // задать фиксированный формат для double
  cout.precision(2);                          // всего цифр, если НЕ ios::fixed
  for (int i = 0; i < count; i++)             // снова проход по данным
    { cout << setw(4); cout << i++)           // номер транзакции
      cout << setw(11); cout << data[i] << endl; }    // значение транзакции
  return 0;
}
```

Если следующая считываемая программой сумма помещается в массив, т. е. условие count == size все еще ложно, значение сохраняется в массиве. Обратите внимание на использование указателя data как имени массива. Оператор data[count++] = amount тот же, что и в предыдущих версиях программы, но смысл его другой. В листинге 6.9 data — это имя массива в стеке, а в листинге 6.10 — имя указателя, ссылающегося на неименованный массив в динамически распределяемой области.

Когда все ячейки динамического массива оказываются заполненными и условие count == size становится истинным, дело принимает интересный оборот. В листинге 6.9 выводится сообщение об ошибке и вводимые данные отбрасываются. В листинге 6.10 делается попытка восстановить работу после переполнения массива, выделить в динамически распределяемой области больше памяти и скопировать существующие данные в новый массив.

Программа не может использовать тот же указатель data для распределения дополнительной памяти. Если data получит адрес нового фрагмента памяти, он потеряет адрес существующих данных. Поэтому для выделения массива в динамически распределяемой области потребовался другой локальный указатель q. Размер нового массива вдвое превышает размер существующего. Удваивание размера — распространенная стратегия управления динамически распределяемой памятью, но можно использовать и другие варианты увеличения размера выделяемой памяти.

```
double q = new double[size*=2];          // получить дополнительную динамически
                                         // распределяемую память
```

Изменение size в операторе, выделяющем память,— нежелательный метод. Сопровождающий приложение программист может легко проглядеть данное действие. Лучше сделать это явно перед операцией new:

```
size = size * 2;
double *q = new double[size];          // удвоение размера массива
if (q == NULL)
  { cout << "Нет памяти \n"; return; }
else
  /* копирование данных в массив, на который указывает q */
```

Обратите внимание, что один и тот же тип указателя double* используется для ссылки на одно значение типа double и на массив значений double. Обычный случай. По типу указателя нельзя сказать, ссылается он на массив или на одно значение. Можно только запомнить, на что он указывает. Это затрудняет пояснение программисту намерений разработчика.

Полезно всегда проверять успешность распределения памяти. Нехватка памяти — редкая ситуация, и программисты часто пренебрегают проверкой возвращаемого операцией new значения. При копировании существующего массива в первую часть нового массива имена указателей можно использовать как имена массивов, избежав тем самым вычислений с указателями.

```
for (int i=0; i < count; i++)
  q[i] = data[i];                    // копировать старые данные в первую
                                     // половину новых данных
```

Некоторые программисты предпочли бы организовать этот цикл, используя арифметические операции с указателем через индексы, например:

```
for (int i=0; i < count; i++)        // копировать старые данные
  *(q+i) = *(data+i);
```

Другим более по вкусу цикл с увеличением указателей для ссылки на следующую ячейку в динамически распределяемой области.

```
for (double *p=q, *r=data; p-q < count; p++, r++)          // годится?
  *p = *r;
```

Третьи используют такую форму:

```
double *p = q, *r = data, int i = 0;
while (i++ < count)
  *p++ = *r++;                                    // правда, красиво?
```

Лично мне кажется, что лучше всего по возможности придерживаться обычных обозначений массивов и избегать арифметических операций с указателями. Между тем, оба вида циклов вполне законны в C++. Вы увидите их в унаследованном коде C/C++.

Но вернемся к динамическому распределению памяти. После копирования имеющихся данных в новый массив исходный массив нужно удалить, а указатель на существующий массив переназначить на новый массив, сделав его следующей версией существующего массива. Важны оба шага и их последовательность. Если не удалить существующий массив, то в программе будет "утечка памяти". Если же сначала переназначить указатель на новый массив, то невозможно будет удалить имеющийся.

```
Введите сумму (или 0 для завершения): 22
Введите сумму (или 0 для завершения): 33
Введите сумму (или 0 для завершения): 44
Введите сумму (или 0 для завершения): 55
Выделено больше памяти: размер = 6
Введите сумму (или 0 для завершения): 66
Введите сумму (или 0 для завершения): 0

Общая сумма 5 значений равна 220

Ном. транз.   Сумма

   1          22.00
   2          33.00
   3          44.00
   4          55.00
   5          66.00
```

Рис. 6.10. *Вывод программы из листинга 6.10 (с отладочными сообщениями)*

Результат выполнения программы из листинга 6.10 представлен на рис. 6.10.

В программе из листинга 6.10 проблем немного. Обратите внимание, что есть только один способ выхода из цикла: когда операции >> не удается получить очередную сумму транзакции. Очень распространена следующая ошибка: программисты часто оставляют в исходном коде уже ненужные операторы. Представьте, как будет гадать сопровождающий приложение программист, что же значит эта проверка.

На самом деле программисту придется догадаться, что данная проверка ничего не означает, а понять это всегда труднее, чем нечто действительно осмысленное.

Еще один пример такой склонности программистов — размещение определения `amount`. Чтобы избежать конфликтов имен, особенно при сопровождении программы, и сделать программу более понятной, важно определять переменные на максимально глубоком уровне в структуре блоков. Вот почему указатель `q` определяется в локальном блоке оператора `if`. С переменными `total`, `data`, `count` этого сделать нельзя — они нужны вне цикла ввода, однако переменную `amount` можно определить внутри данного цикла. Вопрос не так уж существен, поскольку программа маленькая, но выбор подходящего места для определений — важный навык, и в этом следует попрактиковаться.

Завершая работу, наша программа не возвращает оставшуюся память в динамически распределяемую область. Вероятно, в данном случае опасности нет, так как операционная система выполнит "очистку", но это нельзя считать хорошим стилем программирования. И не стоит полагаться на предусмотрительность разработчиков ОС.

Бессмысленно даже обсуждать, опасны ли "утечки памяти". Нужно выработать до автоматизма привычку: при распределении памяти из динамической области выбрать место, где эта память будет возвращаться обратно. В данном случае функции `main()` следовало бы делать это непосредственно перед оператором `return`:

```
delete [] data;          // массив (а не указатель) удаляется
```

Обратите внимание на квадратные скобки в операции `delete` при освобождении занятой массивом памяти. Операция `delete` не настолько интеллектуальна, чтобы понять, на что ссылается указатель — на массив или на отдельное значение. Скобки показывают, что удаляется массив, а не переменная. Здесь также выбран компромисс между интересами программиста и разработчиков компилятора. C++ облегчает жизнь создателям компилятора.

При копировании существующего массива в новый перед освобождением имеющегося массива для ограничения цикла используется переменная `count`, а не `size`, хотя данный сегмент начинается с проверки `count == size`. В момент копирования

они не равны, так как перед распределением массива значение size было увеличено. Возможно, лучше увеличивать size после копирования.

```
if (count == size)                                  // нет памяти, запросить еще
  { double *q = new double[2*size];                 // удвоить размер массива
    cout << "Выделена дополнительная память: размер = " << size << endl;
    for (int i=0; i < size; i++)                    // копирование старых данных
       q [i] = data [i];
    size *= 2;                                  // удвоить размер для следующей проверки
    delete [] data;                             // не забывайте об освобождении памяти
    data = q; }                                 // привязка главного указателя
```

На рис. 6.11 сведены операции для обработки ситуации переполнения массива. Рис. 6.11a демонстрирует выделенный в динамически распределяемой области массив data[] и переменные в стеке — amount, count и size, когда обнаружено переполнение (под data[] понимается ссылка на массив data в динамически распределяемой области). На рис. 6.11b приведен массив q[] в динамически распределяемой области после копирования значений из массива data[] и массив data[] после возврата памяти в динамическую область. На рис. 6.11c демонстрируются оба указателя, data и q, ссылающиеся на новый массив в динамически распределяемой области, а на рис. 6.11d — массив после удаления указателя q в соответствии с правилами области действия.

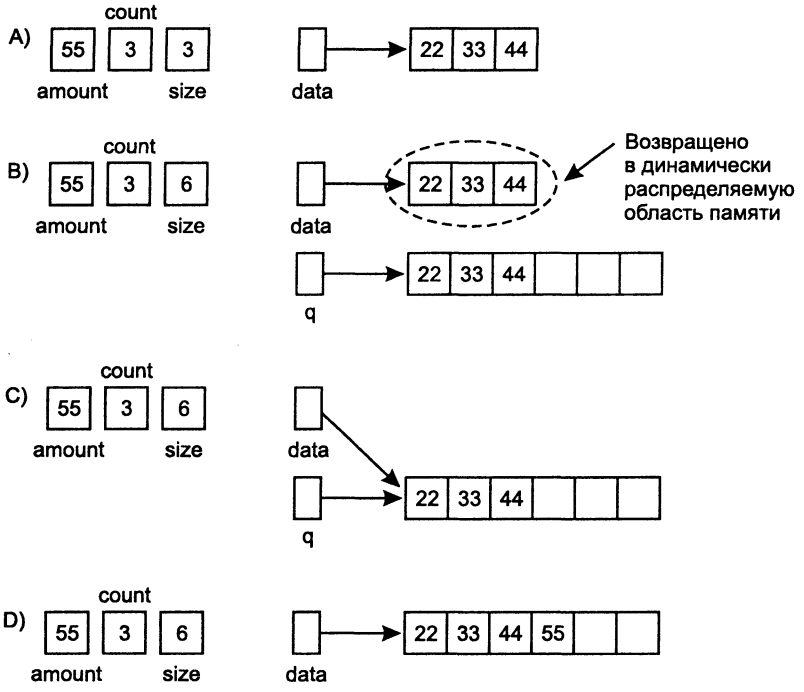


Рис. 6.11. *Последовательность действий при обработке ситуации переполнения массива*

Следующий пример полезного динамического массива связан с вводом текста. Когда программа получает вводимые пользователем символьные данные (например, фамилию заказчика и пр.), то трудно представить, что потребуется массив больше 30–50 символов. А вдруг на клавиатуре залипла клавиша? Данные переполнят этот короткий массив и запортят содержимое памяти. Кроме того, при чтении данных из файла или из телекоммуникационной линии нет никакой гарантии, что размер будет ограничиваться конкретным значением. Риск порчи содержимого памяти всегда есть, как бы ни был велик резервируемый массив фиксированного размера.

В листинге 6.11 показано решение данной проблемы. Идея состоит в том, чтобы вводить данные в относительно короткий именованный массив в стеке (buf[]) и копировать их в массив в динамически распределяемой области (на который ссылается указатель data). Если данные продолжают поступать, они снова считываются в стек (записываются поверх данных, уже скопированных в динамический массив), затем в динамически распределяемой памяти выделяется массив большего размера (указатель temp), в него копируются данные из предыдущего динамического массива (указатель data) и добавляются данные из массива в стеке (buf[]).

Листинг 6.11. Использование динамического массива для бесконечной строки ввода

```
#include <iostream>

using namespace std;

int main(void)
{
  const int LEN = 8; int len = 1;                  // короткий массив для отладки
  char buf[LEN], *data = 0;                        // инициализировать нулем при первом проходе
  cout << "Наберите текст, нажмите Enter: \n";
do {
  cin.get(buf,LEN);                                // данные поступают в массив в стеке
  len += strlen(buf);                              // общая длина старых данных
  char *temp = new char(len);                      // запрос нового динамического массива
  if (temp == 0)                                   // проверка успешности распределения
    { cout << " Нет памяти: программа завершена\n";
      return 0; }                                  // неудача, отказ
  if (data == 0)
    strcpy(temp,buf); }                            // копирование данных из буфера ввода
  else
    {strcpy(temp,data); strcat(temp,buf); }        // копирование данных
    delete [] data;                                // удалить существующий массив
    data = temp;                                   // указатель на новый массив
    cout << " Всего: " << len << добавлено: " << buf << endl;
    cout << Динамический буфер << data << endl;    // отладка
  char ch = cin.peek();                            // что осталось в буфере?
  if (ch == '\n')                                  // выйти, если новая строка
    { ch = cin.get(); break; }
  } while (true)                                   // или продолжать до конца файла
  cout << "\nВы ввели следующую строку: \n\n";
  cout << data << endl;                            // тот же синтаксис, что и для массивов
  delete [] data;
  return 0;
}
```

В листинге 6.11 пользователь вводит данные в массив buf[] в стеке, размер которого для демонстрации работы алгоритма специально выбран небольшим (LEN = 8 символам). Функция get() считывает вводимые данные, пока не получит LEN - 1 символов или не найдет в потоке ввода символ новой строки. (Его следует удалить другими способами, например вызовом функции get(), считывающей ровно один символ.)

В других случаях get() добавляет к строке в buf[] завершающий ноль. Далее программа отводит len символов в динамической области, чтобы вместить считанные из buf[] символы. В первый раз это len = len + strlen(buf), где len содержит 1, а strlen() — число символов без завершающего нуля. Если распределение памяти не удалось (указатель temp установлен в 0), то программа завершает работу.

При первом проходе цикла (указатель data все еще инициализирован нулем) этим все заканчивается: программа копирует введенные данные из buf[] в массив, на который указывает temp. Здесь эквивалентность между указателем и традиционной записью массива оказывается очень удобной. Передается указатель temp на функцию strcpy(), и копируются символы из buf[] в массив, на который указывает temp.

Если это не первый проход цикла (указатель data не ноль, он ссылается на массив, выделенный ранее в динамической области), все усложняется. Сначала программа копирует предыдущие данные во вновь распределенный массив с помощью strcpy(), а затем в конец предыдущих данных вызовом функции strcat() присоединяются символы из buf[].

Теперь указатель temp ссылается на обновленную строку ввода, а указатель data — на предыдущие данные. Далее программа удаляет предыдущие данные и устанавливает указатель data на обновленную строку ввода.

Следующая задача — определить, что произошло при вызове get(). Был ли ввод прекращен из-за обнаружения символа новой строки (который остался в строке ввода) или из-за того, что введено LEN - 1 символов и массив buf[] переполнен? Чтобы выяснить это, программа анализирует следующий введенный символ, вызывая функцию peek(). Если это действительно символ новый строки (что будет иметь место для короткой строки ввода), программа удаляет его, вызывая другую функцию get(), которая считывает один символ и завершает цикл.

Если введенная строка не помещается в буфер ввода, функция get() в начале цикла do завершает ввод после считывания LEN - 1 символов и добавляет в конец массива buf[] нулевой терминатор. Функция peek() будет возвращать следующий символ, отличный от символа новой строки. Не стоит удалять его из потока ввода, ведь это будет первый символ, считываемый при следующем вызове get().

При следующей итерации цикла do оператор get() в начале цикла считывает следующую порцию символов в массив buf[]. На этой итерации нужно скопировать данные из buf[] в динамический массив.

На этот раз массив, на который ссылается символьный указатель data, уже есть. Его длина (включая нулевой завершающий символ) определяется переменной len. Программа использует локальный указатель temp для выделения в динамической области нового массива (с указателем data) достаточного размера для размещения существующего массива в динамически распределяемой области и вновь введенных символов в buf[]. Вот откуда взялось выражение len += strlen(buf). Программа наполняет вновь выделенный массив, копируя в него массив с указателем data и соединяя результат с содержимым массива buf[]. После этого она удаляет имеющийся массив, на который ссылается data, и устанавливает указатель data на вновь распределенный массив (на него указывает temp). Итерации продолжаются, пока следующий вызов peek() не обнаружит символ новой строки или конца файла.

```
Наберите текст, нажмите Enter:
Hello World!
Всего: 8 добавлено: Hello W
Динамический буфер: Hello W
Всего: 13 добавлено: orld!
Динамический буфер: Hello World!

Вы ввели следующую строку:

Hello World!
```

Рис. 6.12. *Вывод программы из листинга 6.11 (с отладочными сообщениями)*

Результат выполнения программы показан на рис. 6.12.

Здесь массивы в динамически распределяемой области памяти также не имеют имен. На них можно ссылаться через указатели, а память для указателей выделяется в стеке. (Поэтому указатели имеют имена temp и data.) Программа использует эти указатели точно так же, как массивы, для которых память выделяется в стеке. Например, указатели temp и data передаются функциям strcpy(), strcat() и strlen() как обычный массив buf[]. Это же относится к операции вставки << в конце листинга 6.11. Указатель data используется как имя массива. Разница в том, что память для именованных массивов возвращается по правилам языка в конце области действия, а динамические массивы освобождаются с помощью явной операции delete (обратите внимание на скобки в операции delete).

На рис. 6.13 показаны операции распределения памяти для ввода данных из рис. 6.12. Рис. 6.13a демонстрирует заполнение массива `buf[]` строкой `"Hello W"`, когда указатель `data` устанавливается в 0. Из рис. 6.13b видно, что переменная `len` содержит 8, `temp` указывает на массив в динамически распределяемой области из 8 символов, а `data` указывает на тот же массив. (Обратите внимание, что операция `delete` с нулевым указателем не дает никакого эффекта.) Рис. 6.13c демонстрирует массив `buf[]` после ввода `"orld!"`. Рис. 6.13d показывает, что `len` содержит 13, `temp` указывает на массив со строкой `"Hello World!"`, массив, на который ссылается `data`, удаляется, а `data` указывает на тот же массив, что и `temp`.

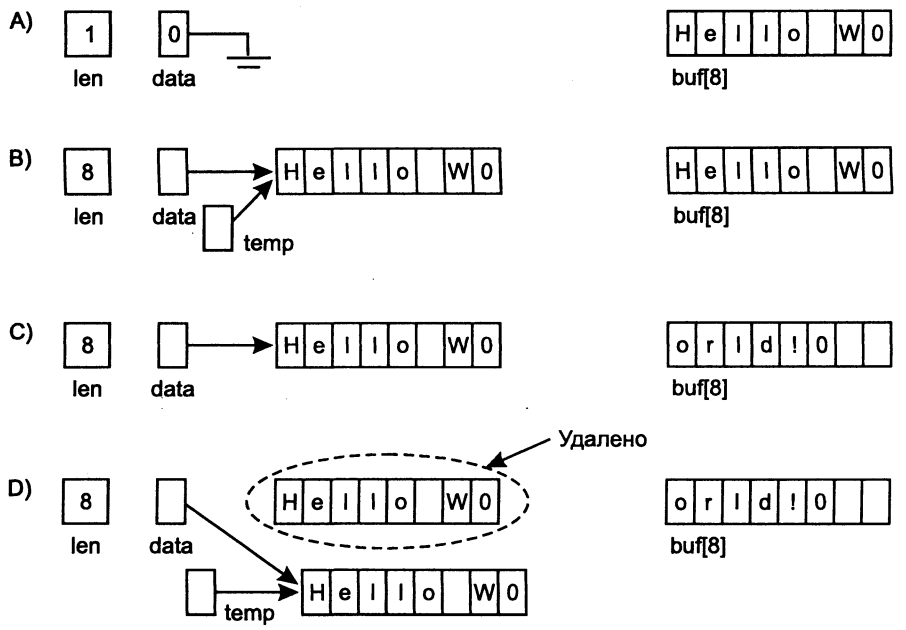


Рис. 6.13. *Диаграмма для указателей при вводе данных в соответствии с рис. 6.12*



Внимание Не разрешается применять операцию `delete` дважды к одному указателю, не инициализируя этот указатель после первой операции `delete`, однако можно применить операцию `delete` к указателю, установленному в 0. Такая операция никакого эффекта не дает.

Если при первом чтении вы чувствуете, что тема слишком сложна, пропустите материал. По мере приобретения опыта управление памятью будет казаться вам все менее сложным. Однако для получения такого опыта нужно поупражняться на простых примерах, составлении диаграмм (подобных представленным в данной главе), развивать навыки отладки и интуицию.

Если материал вам вполне понятен, идите дальше. В предыдущем примере вводилась одна строка данных произвольной длины. В C++ с его распределяемыми в стеке массивами фиксированной длины для работы с такими строками требуется определенное искусство. Аналогичные методы можно применять во многих реальных приложениях.

В следующем примере усложняется предыдущая задача: здесь может считываться любое число строк произвольной длины. Конечно, нет никакого смысла просто так считывать данные с клавиатуры, поэтому показаны также методы сохранения информации в файле на диске.

В листинге 6.12 приведен алгоритм, реализованный в листинге 6.11,— он используется здесь как внутренний цикл. Внешний цикл продолжает чтение данных, пока пользователь не нажмет Enter, не набрав в строке никаких данных. Такая пустая строка служит контрольным значением, завершающим ввод.

Листинг 6.12. Использование динамического массива для ввода произвольного набора строк

```
#include <iostream>

using namespace std;

int main(void)
{
  const int LEN = 8; char buf[LEN];
  int cnt = 0;
  cout << "Наберите текст (или просто нажмите Enter, чтобы закончить ввод): \n";
do {                                                    // начать внешний цикл для ввода строк
  char *data = new char[1]; data[0] = 0;                // первоначально пусто
  int len = 0;                                          // сначала размер равен 0
  do {                                                  // начало внутреннего цикла для
                                                        // сегментов строк
    cin.get(buf,LEN);                                   // данные поступают в массив в стеке
    len += strlen(buf);                                 // общая длина старых данных
    char *temp = new char(len+1);
    strcpy(temp,data); strcat(temp,buf);
    delete data;
    data = temp;                                        // расширение длины строки
    cout << "Выделено << len+1 << ": " << data << endl;
    char ch = cin.peek();                               // что осталось в буфере?
         if (ch == '\n' || ch == EOF)                   // выход, если новая строка
            {  ch = cin.get();                          // удалить из ввода
               break; }
    } while (true);
    if (len == 0) break;                                // завершение на пустой строке
    cout << " строка " << ++cnt << ": " << data << endl;
    delete [] data;
  } while (true);                                       // продолжать до пустой строки
  return 0;
}
```

Между программами в листинге 6.11 и 6.12 есть несколько интересных различий. В листинге 6.11 переменная len обозначает размер массива, выделенного в динамической области. В листинге 6.12 переменная len обозначает число символов, копируемых в массив в динамически распределяемой области. Размер массива на единицу больше, чтобы вместить нулевой терминатор.

Эти две программы обрабатывают первое чтение по-разному. В них различается первое чтение в buf[] и другие операции чтения. При первом чтении массив в динамически распределяемой области еще не существует, поэтому размер памяти для запроса на 1 больше, чем размер считанных в buf[] символов, а не сумма символов в массиве динамически распределяемой области и в buf[]. Это означает, что нужно использовать оператор if.

```
if (data == 0)
  len = strlen(buf) + 1;          // первый раз копируется только из buf[]
else                            // в противном случае копируется из data[] и buf[]
  sen = strlen(data)+strlen(buf)+1;
```

Кроме того, при первом чтении массив в динамически распределяемой области принимает данные только из buf[]. Во время других итераций во вновь выделенный в динамической области массив копируются данные из существующего динамического массива и из массива buf[]. Вот почему листинг 6.11 содержит оператор if:

```
if (data == 0)
  strcpy(temp,buf);          // первый раз копировать только из buf[]
else                         // в противном случае копируется из data[] и buf[]
  { strcpy(temp,data), strcat(temp,buf); }
```

Листинг 6.11 не содержит первых операторов if. Нередко программисты чувствуют, что дополнительные проверки усложняют программу и пытаются избежать их путем более разумного использования данных при обработке различных ситуаций. В листинге 6.11 data инициализируется нулем, а len — единицей. Следовательно, можно использовать оба случая в следующем операторе:

```
len = len + strlen(buf);          // работает для первого и следующего чтения
```

Тестирование программы показывает, что она работает, но все равно остается некоторое беспокойство. Данный оператор нуждается в пояснении (и тщательной проверке). Расположенный выше оператор if говорит сам за себя. Что предпочтительнее: понятный с первого взгляда объемный исходный код или более компактный вариант, нуждающийся в дополнительных пояснениях? В предыдущих главах говорилось одно, но теперь я хочу сказать о другом.

В листинге 6.12 нужно отметить еще один момент. Указатель data первоначально ссылается на массив в динамически распределяемой области, у которого размер равен 1. Массив этот содержит единственный символ — нулевой терминатор (обычно такую строку называют пустой). Переменная len инициализируется значением 0 — длиной данной пустой строки:

```
int len = 0;                                    // начальная длина данных
char *data = new char[1]; data[0] = '\0';       // пустая строка
```

В то же время нет никакой разницы между первой итерацией и всеми остальными. В программе добавляется длина buf[] к длине data[], затем data[] копируется в новый массив в динамически распределяемой области (сначала это пустая строка, а потом к ней добавляется содержимое buf[]:

```
do {                                        // начало внутреннего цикла
  cin.get(buf,LEN);                         // получить следующий сегмент строки
  len += strlen(buf);                       // обновить общую длину строки
  char *temp = new char[len+1];             // выделить новый массив в heap
  strcpy(temp,data); strcat(temp,buf);          // слить данные
```

Неплохой код, но, честно говоря, вариант с двумя операторами if более понятен.

Еще один вопрос — завершение работы программы. Если пользователь нажимает Enter (Return), программа из листинга 6.12 должна заканчивать работу. Предполагается, что при этом вводится символ новой строки '\n' (ASCII 10), а вызов функции peek() принимает его и завершает внутренний цикл. Проверка len == 0 прерывает внешний цикл. Но только не на моей машине. Когда я ввожу символы и нажимаю Enter, вводится символ новой строки, но если просто нажать Enter, вводится символ "конца файла" (EOF). Вот поэтому в листинге 6.12 проверяется как символ новой строки, так и EOF (константа со значением –1):

```
if (ch == '\n' || ch == EOF)          // выход, если новая строка или EOF
  { ch = cin.get(); break; }          // но сначала удалить его из буфера ввода
```

Кстати, программа из листинга 6.11 этого не делает, а следовательно, при нажатии Enter без набора символов, она входит в бесконечный цикл, не встречая символов новой строки, удовлетворяющих условию if. Такая хорошая программа — и такая ошибка! Обидно.

Но программа из листинга 6.12 не многим лучше. Конечно, она работает, и не зацикливается (как кажется), однако тип переменной ch в ней определен как char, а затем эта переменная сравнивается с EOF (отрицательным значением). Такое сравнение будет работать только в том случае, если char по умолчанию определяется как signed (переменная со знаком). На моей машине это так, но на другой необязательно. Замена определения ch на следующее позволит посмотреть, что происходит в этом случае:

```
unsigned char ch = cin.peek();          // конец файла, 255
```

```
Наберите текст (или просто нажмите Enter,
чтобы закончить ввод): First line
Выделено 8: First l
Выделено 11: First line
 строка 1: First line
This is the last line
Выделено 8: This is
Выделено 15: This is the la
Выделено 22: This is the last line
 строка 2: This is the last line

Выделено 1:
```

Рис. 6.14. *Вывод программы из листинга 6.12 (с отладочными сообщениями)*

Запустив программу из листинга 6.12, можно видеть, что она действительно зацикливается.

Это общая проблема переносимости. Хорошим решением будет использование типа int:

```
int ch = cin.peek();          // конец файла, -1
```

Для программы из листинга 6.12 этого достаточно. Выполнение программы показано на рис. 6.14.

Итак, примеры становятся все более сложными, настало время подумать о других методах управления памятью. Если данная тема кажется вам слишком сложной, переходите сразу к главе 7, где рассказывается о функциях C++, но не забудьте вернуться сюда и прочитать разделы, посвященные динамическим структурам.

Динамические структуры

В предыдущем разделе рассматривалось применение динамически распределяемой памяти для массивов, что позволяет определять размеры массивов на этапе выполнения, а не на этапе компиляции. Такой метод работы с указателями намного полезнее, чем методы, обсуждавшиеся в начале главы (ссылки на переменные в стеке или распределение в динамической памяти отдельных переменных).

Применение динамических массивов устраняет опасность порчи содержимого памяти или непроизводительного расходования доступного пространства. За это приходится расплачиваться дополнительной сложностью программы и возможностью "утечки памяти", если программист работает с ее распределением некорректно. Следует принимать во внимание и производительность программы. Управление динамически распределяемой памятью требует времени. Для большинства приложений такое влияние на производительность несущественно, но если память выделяется и освобождается слишком часто, возможно снижение быстродействия программы.

Для всех массивов (фиксированного размера и динамических) характерно, что добавление и удаление элементов выполняется быстро и просто, только когда это последний действительный элемент массива. Если элементы добавляются или удаляются в середине массива, то все происходит гораздо сложнее и медленнее. Динамически распределяемые структуры — хорошая альтернатива применению массивов с часто включаемыми и удаляемыми элементами.

Определяемые программистом структуры можно распределять как отдельные узлы и соединять в связанные списки (сети узлов). Чтобы можно было включить узел в связный список, он должен представлять собой структуру, содержащую по крайней мере два компонента: информационный элемент и адрес следующего узла (указатель на следующий узел в связанном списке). Информационный элемент может быть отдельным значением или структурой с несколькими полями (в соответствии с требованиями приложения). Чтобы сконцентрироваться на вопросах программирования, а не на деталях приложения, будем рассматривать очень простую структуру, в которой информационный элемент содержит только одно значение (например, сумму транзакции).

При определении типа узла можно свободно выбирать имя поля с адресом следующего узла. Назовем его next. А вот его тип свободно выбрать нельзя:

```
struct Node {
  double amount;                    // информационный элемент
  Node* next; }                     // связь со следующим узлом
```

Какое бы имя типа для узла ни использовалось, имя типа поля next будет таким же — добавляется лишь обозначение указателя, поскольку поле next представляет собой указатель на структуру типа Node. Хорошо бы также использовать один и тот же тип узла в разных контекстах, при необходимости меняя тип информационного поля. Один из способов сделать это — ввести другой тип Item и определить его с помощью typedef. (Другой способ заключается в применении шаблонов C++ — они более гибкие, но при этом сложнее.)

```
typedef double Item;                // Item - синоним double
struct Node {
  Item item;                        // информационный элемент
  Node* next; } ;                   // ссылка на следующий узел
```

Выделять узлы в динамической области можно в любое время. Идея использования узлов состоит в создании узла только тогда, когда программе нужно сохранить информацию в новом элементе (например, после чтения данных с клавиатуры, из файла или из сети). Следовательно, нет необходимости резервировать память заранее, т. е. нет нужды в массивах, а потому отсутствует опасность переполнения и непроизводительного расходования памяти, не нужно сдвигать элементы для вставки или после удаления.

Динамическое управление памятью со связанными узлами сложнее, чем при использовании динамических массивов, но это очень важный навык программирования. Указатели, применяемые для операций с узлами, являются именованными переменными. Память для них выделятся в стеке, как для глобальных или локальных (в некоей области действия — функции или блоке) переменных. Указатели должны: правильно определяться, инициализироваться и обрабатываться.

Программистов редко интересует значение самого адреса. Указатель используется не для определения хранящегося в нем адреса, а для доступа к объекту, на который данный указатель ссылается. Для этого имя указателя применяется с именем объекта или без него (поскольку распределяемый в динамической области объект имени не имеет).

Следующий фрагмент программы показывает типичную ошибку программирования: здесь определяются два указателя-переменных, которые не инициализируются:

```
Node *p, *q;                        // область действия * - одно имя
q->item = amount;                   // портит содержимое ячейки,
                                    // на которую ссылается q
```

Неинициализированный указатель может ссылаться на что угодно. Если программа не использует данную область памяти, то результаты могут быть корректными. Если эта область задействована ОС или другой программой, возможны проблемы.

Нельзя путать терминологию или обозначения. У программиста на это должна быть своего рода интуиция. До сих пор нам приходилось иметь дело только с нуждающимися в инициализации r-значениями. В следующем примере с целыми ясно, что переменную x перед использованием в присваивании нужно инициализировать (как r-значение), но переменная y в инициализации не нуждается, поскольку она применяется как l-значение:

```
int x; int y;                       // определение неинициализируемых переменных
y = x;                              // x требует инициализации, y - нет
```

В приведенном выше примере q->item используется как l-значение и не требует предварительной инициализации, однако q применяется как r-значение и должно что-то содержать перед доступом к полю item.

Когда указатель (например, q) правильно инициализируется, его содержимым будет адрес структуры типа Node. Неизвестно, что именно содержит q: адрес поля item или поля next, начало структуры или что-то среднее. Не стоит пытаться выяснить это и использовать для оптимизации программы. Адрес указателя должен оставаться абстракцией адреса. Каково бы ни было содержимое q, *q есть значение, на которое ссылается указатель. В данном случае — неименованная структура типа Node. Доступ к данному значению называется *разыменованием указателя*. Аналогично, q->item — значение поля next в той же структуре. Доступ к полям неименованной структуры через указатель, ссылающийся на эту структуру (q->item и q->next) называется разыменованием указателя.

Некоторые программисты не любят работать с двумя операторами указания — стрелкой и звездочкой. Можно использовать и единообразную запись: (*q).item вместо q->item и (*q).next вместо q->next. Скобки здесь необходимы, так как операция указания имеет более высокий приоритет, чем операция разыменования. Следовательно, *q.item означает *(q.next), а это синтаксическая ошибка — операция указания может применяться здесь только к структурной переменной, а не к указателю.

Таким образом, программа не должна разыменовывать неинициализированный указатель. Если указатель глобальный, то по умолчанию он имеет значение NULL, и разыменование такого указателя даст ошибку этапа выполнения (обычно программа прекращает работу). Если указатель локальный, его значением будут просто произвольные данные. Интерпретируемое как адрес, такое значение может указывать на любое место в памяти (в динамически распределяемой области или нет). Нужно следить за тем, чтобы не запортить содержимое памяти из-за получения подобных некорректных значений.

Установить значение указателя можно несколькими способами. Один из них состоит в присваивании ему адреса именованной переменной с помощью операции получения адреса q = &count, но это не особенно полезный способ. Остаются еще два способа присваивания значения указателю:

- Выделение новой неименованной переменной в динамически распределяемой области и присваивание указателю значения, возвращаемого операцией new (в действительности, неизвестно, на что она ссылается — на начало или конец динамически распределяемой памяти).
- Поиск указателя, который уже ссылается на интересующую нас область памяти, и использование его как источника в присваивании. Этот указатель может быть переменной в стеке или полем переменной в динамически распределяемой области.

Это все, что касается инициализации и присваивания указателей. В следующем фрагменте применяются оба метода:

```
Node *p, *q = new Node;          // q инициализирован, а p - нет
q->item = amount;                // сохраняет значение amount
                                 // в динамической области
q->next = NULL;                  // популярное контрольное значение
                                 // для связанных списков
p = q;                           // p ссылается на тот же узел, что и q
```

Во многих алгоритмах возникает необходимость перебирать связанную структуру, т. е. проходить каждый узел и выполнять некоторые операции (получение значения элемента, проверка достижения последнего узла и т. д.). Один из способов сделать это состоит в ведении счетчика узлов (подобно массиву). Другой —

в переборе узлов до обнаружения контрольного значения в списке. Обычно в качестве контрольного значения используется содержимое поля next последнего узла, которому присваивается NULL. Преимущество такого подхода в том, что данное значение нельзя спутать с другими возможными значениями указателя. Как уже говорилось выше, обычный 0 для этого также подходит, но многие программисты предпочитают применять определенное в библиотеке значение NULL, указывая тем самым на то, что программа имеет дело с указателями.

C++ не позволяет присваивать адрес переменной одного типа указателю другого типа. В этом смысле C++ — язык со строгим контролем типов. В следующем примере исходного кода программист пытается вывести содержимое каждого байта переменной node (на которую ссылается указатель q) в виде символов ASCII. Обратите внимание, что не все содержимое двоичных полей Node может выводиться, как отображаемые символы. Именно такого рода злоупотребления и не допускает строгий контроль типов:

```
char *c = q;                                    // нет, это синтаксическая ошибка
for (int i = 0; i < sizeof(Node); i++)          // проход каждого байта
  cout << *c++ << ' ';                          // вывод каждого байта как символа
```

C++ позволяет делать то, что, по мнению программиста, должно быть сделано. Ведь это свободная страна, в конце концов. Если требуется распечатать каждый байт структуры, то такое возможно, следует лишь указать компилятору (и сопровождающему приложение программисту), что применяется другой тип указателя и что вы знаете, что делаете). Механизм уже известен — приведение типа в C++. Вот пример:

```
char c = char*) q;                              // нет, это синтаксическая ошибка
for (int i = 0, i < sizeof(Node); i++)          // перебор каждого байта
  cout << (int) (*c++) << ' ';                  // вывод каждого байта как символа
```

Обратите внимание, что типы char и Node несовместимы. Значение одного типа нельзя преобразовать в значение другого даже с помощью приведения типа. Это еще один пример строгого контроля типов в C++. Значения указателей разных типов нельзя присваивать друг другу непосредственно, но можно преобразовать их через приведение типа. Почувствуйте разницу. И не злоупотребляйте преобразованием указателей.

Если программа формирует связанную структуру (в цикле), каждый узел создается в динамически распределяемой памяти, заполняется данными (вводимыми с клавиатуры или из файла) и добавляется к связанной структуре. Есть несколько видов связанных структур. Мы рассмотрим простой связанный список, в котором новый узел добавляется к концу списка.

При наличии такой структуры, как связанный список, программа может поочередно обращаться к каждому узлу, начиная с первого и до узла, содержащего в поле next контрольное значение. Проблема в том, как добраться до конца списка при включении новых узлов. Перебор каждого узла от начала до контрольного значения — дело слишком сложное и дорогое в смысле времени выполнения, особенно, если список разрастается в размере.

Одно из решений данной проблемы состоит во введении указателя на последний узел списка. При создании нового узла он добавляется к списку без перебора других узлов. Что означает в этом случае "добавление"? То, что поле next последнего узла (которое содержало адрес NULL) будет ссылаться на новый последний узел. Следовательно, нужно найти имена для поля next последнего узла (l-значение в присваивании) и адрес нового узла (r-значение в присваивании). Между тем оба узла распределяются в динамической области — имен они не имеют! Это означает, что нужно найти указатели, ссылающиеся на эти два узла (последний и новый). В следующем фрагменте указатель, ссылающийся на последний узел, называется last, а имя указателя, ссылающегося на новый узел — q. Таким

образом, присваиванием, добавляющим новый узел в конце связанного списка, будет last->next = q. В контексте это выглядит так:

```
Node *last;                          // указатель на последний узел
do {                                 // до EOF
    ...
  Node* q = new Node;                // создает новый узел в динамической области
  if (q == 0)                        // проверка на успешное выполнение запроса
    {  cout << "Нет памяти: ввод прекращен" << endl;
       break; }                      // корректно завершить, если неудача
  q->item = amount;                  // заполнить узел данными программы
  q->next = NULL;                    // контрольное значение для конца списка
  last->next = q;                    // добавить как последний узел в список
  ...                                // другие необходимые операции
} while (true);
```

Хорошее решение. Оно показывает, как можно быстро добавить к связанному списку новый узел, не перебирая в списке все имеющиеся узлы. Не указаны лишь две важные вещи: как начать и как закончить, т. е. как подсоединить к пустому списку самый первый узел и как убедиться, что на следующей итерации цикла указатель последнего узла действительно ссылается на последний узел в списке, а не на бывший последний (предшествующий вновь добавленному).

При добавлении к списку самого первого узла выражение last > next не имеет смысла, так как в списке нет узлов и, следовательно, next не может принадлежать выделенному ранее узлу. Это означает, что при добавлении к списку самого первого узла нужно обойти данное присваивание и делать что-то еще. Например, присоединить первый узел к "голове" списка.

Обычно "голова" списка просто представляет собой еще один указатель. Назовем его data. Один из способов сообщить об отсутствии присоединенных к списку узлов состоит в ведении счетчика узлов списка. Когда счетчик count равен 0, новый узел нужно добавлять к указателю списка data. Когда счетчик не равен 0, новый узел следует добавлять к концу списка, т. е. list->next:

```
Node *last, *data; int count=0     // указатель первый/последний,
                                   // счетчик узлов
do {                               // до конца данных
    ...                            // получение значения amount
  Node* q = new Node;              // создание нового узла в динамической области
  if (q == 0)                      // проверка на успешное выполнение запроса
    {  cout << "Нет памяти: ввод прекращен" << endl;
       break; }                    // корректно завершить, если неудача
  q->item = amount;                // заполнить узел данными программы
  q->next = NULL;                  // контрольное значение для конца списка
  if (count == 0)                  // только для первого узла
    data = q;                      // добавить первый узел в список
  else
    last->next = q;                // добавить как последний узел в список
  ...                              // другие необходимые операции
} while (true);
```

Помните об условном операторе? Это как раз та ситуация, когда данный оператор очень удобен. В зависимости от значения count, выражение возвращает data или last->next. Затем указатель q присваивается соответственно либо указателю data, либо las ->next.

```
(count == 0 ? data : last->next) = q;          // хорошая запись
```

Еще один способ начать список состоит в инициализации указателя списка data значением NULL. В цикле после распределения и инициализации нового узла можно проверить, содержит ли указатель списка NULL. Если да, то новый узел — это первый узел в списке, который следует присоединить к data. Если он не NULL, то новый узел не является первым и его следует добавить к list->next.

```
if (data == NULL)          // это значит, что узлов еще нет
  data = q;                // устанавливает указатель списка на первый узел
else
  last->next = q;          // новый узел добавляется к последнему узлу списка
```

Если предпочитаете условный оператор, можно использовать его:

```
(data == 0 ? data : last->next) = q;          // эффектный код
```

Все это проиллюстрировано на рис. 6.15. На рис. 6.15a — начальное состояние списка, когда указатель data инициализирован значением 0, а указатель last пока может ссылаться на что угодно. Рис. 6.15b демонстрирует состояние списка после добавления первого узла (когда значение amount равно 22). Инициализируется новый узел, на него ссылается указатель q, а указатели data и last ссылаются на следующий узел. Обратите внимание, что next имеет тот же размер, что и указатели data и last, поскольку все они одного типа — Node*. Из рис. 6.15c видно, как выглядит список после добавления еще одного узла (на него ссылается указатель q). Рис. 6.15d демонстрирует первый шаг добавления к концу списка: поле next последнего узла (last->next) устанавливается на новый узел (куда ссылается q). На рис. 6.15e показан второй шаг добавления: указатель last устанавливается на новый узел.

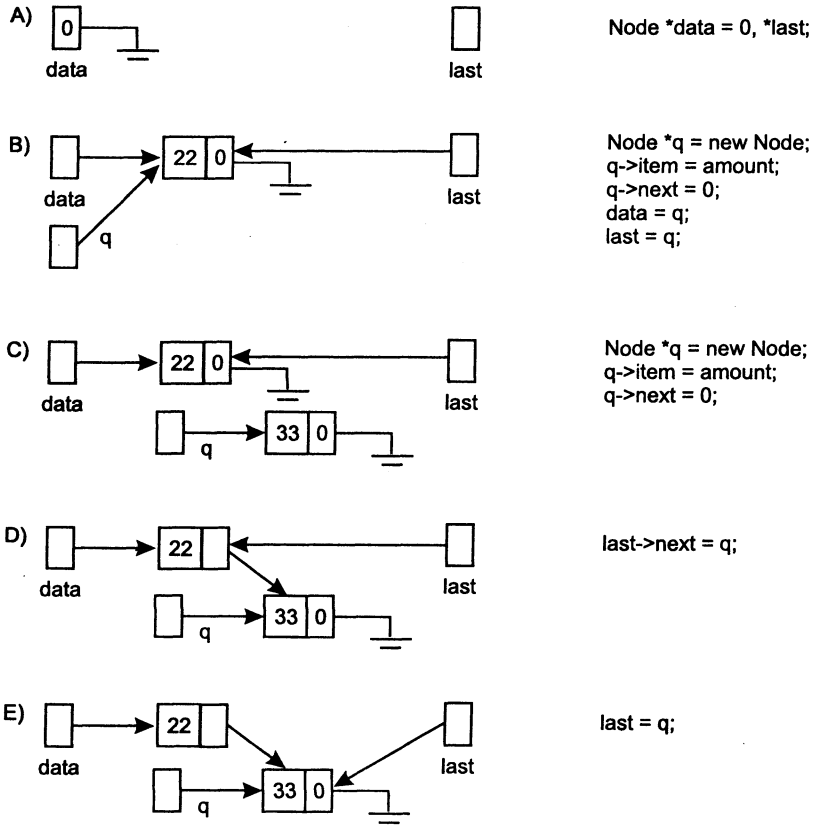


Рис. 6.15. *Диаграмма указателей для включения нового узла в конец связанного списка*

Как видно, после добавления нового узла в конце списка следует переместить указатель last, так как он ссылается на узел, предшествующий последнему, и присваивание last->next на следующей итерации было бы некорректно. Для перемещения указателя last нужно записать оператор присваивания, где указатель last находится слева. Что же будет справа? Чтобы ответить на данный вопрос, нужно найти указатель, уже ссылающийся на цель присваивания, т. е. указатель на новый узел.

Посмотрите на рис. 6.15d. Ссылаются ли какие-нибудь указатели на вновь добавленный узел? Да. Даже два. Один из них — указатель q, который использовался для распределения нового узла. Другой — указатель last->next, который использовался для добавления этого узла в список. Подойдет любой.

```
last = q;                    // переустановить указатель снова на последний узел
```

Используя второй указатель для ссылки на новый узел, можно получить:

```
last = last->next;           // перемещение указателя к следующему узлу списка
```

Это вторая форма перемещения указателя last на самом деле представляет собой обобщенный метод установки указателя на следующий узел в связанном списке. Такой метод очень популярен в алгоритмах обработки списков. Он эквивалентен оператору i++ при переборе элементов массива, перемещающему индекс на следующий элемент.

В листинге 6.13 приведена программа, аналогичная приведенной в листингах 6.9 и 6.10. Данные транзакций считываются с клавиатуры. Вместо выделения фиксированного массива в стеке (как в листинге 6.9) или распределения массива в динамической области памяти программа выделяет память для каждого узла и каждого получаемого значения. Затем узел добавляется в конец связанного списка.

Листинг 6.13. Использование связанного списка узлов в динамически распределяемой памяти

```
#include <iostream>
#include <iomanip>
using namespace std;
typedef double Item;
struct Node {
   Item item;
   Node* next; } ;

int main ()
{ int count = 0;                              // счетчик amount
  Node *data=0, *last;                        // указатели на начало и конец списка

do {                                          // пока не встретится EOF
   double amount;                             // локальная переменная для ввода
   cout << " Введите сумму (или 0 для завершения): ";
   if (amount == 0) break;
   cin >> amount;                             // получить следующее значение double
   if (amount==0) break;                      // остановить ввод, если более нет данных
   Node* q = new Node;                        // создать в динамической области новый узел
   if (q == 0)                                // проверка на успешное выполнение запроса
     { cout << "Нет памяти в динамической области" << endl; break; }
   q->item = amount;                          // заполнить узел данными программы
   q->next = NULL;                            // контрольное значение для конца списка
   (data == 0 ? data : last->next) = q;
   last = q;                                  // last=last->next; также годится
   count++;
     } while (true);
```

```
    cout << "\nВсего загружено " << count << " значений\n";
    if (count == 0) return 0;                          // нет вывода, если нет ввода из файла
    cout << "\nНомер Сумма Промежуточный итог\n\n";    // печать заголовка
    cout.setf(ios::fixed);                             // фиксированный формат для double
    cout.presicion(2);                                 // цифры после десятичной точки
    double total = 0;                                  // сумма для ввода значений
    Node *q = data;                                    // старт в начале списка
      for (int i = 0; i < count; i++)                  // перебор данных списка
           {  total += q->item;                        // накопление итоговой суммы
              cout.width(3); cout << i+1;              // номер транзакции
              cout.width(10); cout << q->item;         // значение транзакции
              cout.width(11); cout << total << endl;   // текущий итог
              q = q->next; }                           // перемещение указателя на след. узел
      Node *p = data, *r = data;                       // инициализация перебора указателей
      while (p != NULL)                                // пока не кончится список
           {  p = p->next;                             // предотвращение "зависания" последнего узла
              delete r; r = p; }                       // удаление узла, переход к следующему
    return 0;
}
```

После чтения всех данных программа перебирает связанный список. Для каждого узла она выводит сумму транзакции и текущий промежуточный итог. Вывод программы представлен на рис. 6.16.

```
Введите сумму (или 0 для завершения): 22
Введите сумму (или 0 для завершения): 33
Введите сумму (или 0 для завершения): 44
Введите сумму (или 0 для завершения): 55
Введите сумму (или 0 для завершения): 66
Введите сумму (или 0 для завершения): 0

Всего загружено 5 значений

Номер    Сумма    Промежуточный итог

  1      22.00            22.00
  2      33.00            55.00
  3      44.00            99.00
  4      55.00           154.00
  5      66.00           220.00
```

Рис. 6.16. *Вывод программы из листинга 6.13*

Код прохода по связанному списку инициализирует локальный указатель q, ссылающийся на начало списка (q = data). Затем он делает count шагов по списку и на каждом шаге обращается к узлу по указателю q (в данном случае накапливает total, выводит Number, Amount и Subtotal). После этого установкой q в q->next выполняется переход к следующему узлу. Когда q становится равным NULL, найден последний узел (его поле next содержит NULL) и цикл завершается.

Еще одна форма этого цикла для перебора списка переустанавливает указатель в заголовке цикла for:

```
double total = 0;                                  // сумма для вводимых величин
int i = 0;                                         // старт в начале списка
for (Node* q=data; q!=NULL; q=q->next)             // проход по списку
  {  total += q->item;                             // накопление total
     cout.width(3); cout << i+1;                   // номер транзакции
     cout.width(10); cout << q->item;              // значение транзакции
     cout.width(11); cout << total << endl;        // текущий итог
     i++; }                                        // увеличение счетчика
                                                   // обработанных узлов
```

Обратите внимание, что имя q уже было ранее в программе. Оно было локальным в цикле ввода, а потому допускается повторное использование данного имени в программе без какого-либо анализа. Если бы оно определялось в области действия функции main(), подобно data, то для дальнейшего применения данного имени нужно было бы анализировать, можно ли использовать его для других целей, или потребуется другое имя. Это еще один пример уменьшения зависимости между отдельными частями программы за счет корректного применения области действия. Тем самым уменьшается сложность программы и упрощается ее сопровождение.

Последний цикл в программе из листинга 6.13 показывает еще один вид прохода по списку. Здесь цель состоит в возврате списка узлов в динамически распределяемую область, чтобы избежать "утечки памяти". В данном простом примере, когда программа распределяет узлы, поочередно перебирая их, а потом завершает работу, это не так важно. О динамически распределяемой памяти позаботится ОС. Это важно для программ, где выделение и освобождение памяти для узлов происходит многократно в процессе выполнения (иногда в течение очень долгого времени). Ясно, что в таких программах неосвобождение выделяемой для узлов памяти приведет к проблемам.

Следующий цикл включен в программу для демонстрации еще одного способа перебора списка. Цикл должен проходить каждый узел списка и удалять его. Здесь также нужно инициализировать указатель для ссылки на первый узел и перемещения к следующему узлу. При достижении последнего узла переход к следующему узлу приведет к тому, что указатель примет значение NULL. Популярным вариантом перебора списка является применение цикла for:

```
for (Node *q = data; q != NULL; q = q->next)          // просмотр каждого узла
  { delete q; }                                       // освобождение памяти в
                                                // динамически распределяемой области
```

Хороший цикл. Его заголовок почти стандартен и может применяться во многих контекстах. Проблема здесь в том, что выражение инкремента q = q->next выполняется после тела цикла и перед проверкой условия завершения цикла. Между тем в теле цикла освобождается память, на которую ссылается указатель q. Эта память может использоваться для других целей, и данная программа никоим образом не должна на нее ссылаться. После delete q данный цикл ссылается на q->next.

Кстати, это вовсе не означает, что программа выполняется некорректно. На одной из моих машин она работала правильно, так как память, на которую ссылался указатель q, была помечена как свободная и не использовалась для других целей. Поэтому выражение q->next действительно давало адрес следующего узла правильно. Однако не нужно полагаться на это и считать допустимым ссылки на чью-то память. На другой машине данная программа аварийно завершалась. Так что корректность результатов программы C++ вовсе не означает, что программа работает правильно.

Программа из листинга 6.13 использует более сложную, но более надежную форму цикла. Указатели p и r ссылаются здесь на один узел. Затем p перемещается на следующий узел, а узел, на который ссылается r, удаляется. После этого p и r снова ссылаются на один узел:

```
Node *p = data, *r = data;      // инициализация перебора указателей
while (p != NULL)               // пока не кончится список
  { p = p->next;                // предотвращение "зависания" последнего узла
    delete r; r = p; }          // удаление узла, переход к следующему
```

Обратите внимание, что "delete r" здесь следует читать как "удален узел, на который ссылается r", а не "узел r удален". Не стоит верить в то, что удаляется сам указатель. Это весьма далеко от истины. Указатель имеет имя, и, следовательно, память для него выделяется в стеке. Таким образом, он удаляется согласно правилам языка при достижении закрывающей фигурной скобки области действия, где указатель определен. Операция delete удаляет только неименованные переменные в динамически распределяемой области.

И еще одно, последнее предупреждение. Когда данная программа работает со связанными узлами в динамически распределяемой области памяти, ее корректная компиляция и правильное выполнение для всех наборов данных не означает, что программа действительно корректна. Это справедливо лишь для алгоритмов, не работающих с динамически распределяемой памятью. Поэтому никогда не стоит применять более сложных решений, чем того действительно требует приложение.

Обмен данными с файлами на диске

Во всех предыдущих примерах программа считывала данные с клавиатуры и выводила их на экран. Это позволяло нам сконцентрироваться на других моментах, но в реальных приложениях все иначе. Программа должна иметь возможность получать данные, созданные другими приложениями, и сохранять результаты для будущего использования. В этом разделе кратко рассматриваются файлы — как еще один способ работы с большими наборами данных.

Как и другие современные языки, C++ не имеет встроенных операций ввода-вывода. Они вынесены из самого языка в библиотеку. Программы C++ могут использовать две библиотеки: стандартную библиотеку ввода-вывода `stdio`, унаследованную из языка C, и более новую библиотеку `iostream`, разработанную специально для C++.

Обе библиотеки поддерживают файловый ввод и вывод. Библиотека C сложна и порождает ошибки.

Это важно знать программистам, поддерживающим унаследованные программы на C. Библиотека C++ способствуют снижению количества ошибок в программах, но более сложна и громоздка. Часто одно и то же действие можно выполнить двумя способами. Чтобы понять, как работает библиотека C++, нужно знать об использовании классов C++, о наследовании, множественном наследовании и другие концепции, которые пока не обсуждались. Вот почему в данном разделе описываются лишь базовые средства, позволяющие считывать данные из файлов на диске и записывать их в файл.

Вывод в файл

Начнем с записи в файл, поскольку это в чем-то проще, чем чтение из файла.

На самом деле запись данных в файл на диске аналогична выводу данных на экран монитора, но вместо предопределенного объекта `cout` используется определяемый программистом объект библиотечного класса `ofstream` (output file stream — поток вывода в файл). Этот класс определен в библиотечном заголовочном файле `fstream`, который нужно включить в исходный код.

Как говорилось в главе 2, объект представляет собой экземпляр класса, комбинирующий данные и поведение, т. е. структуру, компоненты которой включают в себя функции. Библиотечный класс `ofstream` сконструирован так, что все доступные для стандартного объекта `cout` функции доступны и для определяемых программистом объектов класса `ofstream`.

Это очень удобно. Все, что нужно сделать,— направить вывод программы вместо экрана в файл на диске, для чего определяется объект класса `ofstream`, подставляемый в программе вместо объекта `cout`. Операторы вывода (включая операторы форматирования) выполняют те же действия, что и для объекта `cout`. Они преобразуют последовательности битов в переменных программы в последовательности записываемых символов, но выводятся они не на экран, а в файл на диске.

В листинге 6.14 показана программа из листинга 6.12, считывающая набор строк произвольной длины и сохраняющая их в файле `data.out`. Для этого потребовались минимальные изменения.

Как видно, здесь определяется объект `f` класса `ofstream`. В качестве аргумента при создании файлового объекта задается имя физического файла на диске:

```
ofstream f("data.out");              // открыть файл для вывода
```

Этот оператор связывает объект `f` с физическим файлом `data.out` в том же каталоге, что и выполняемый файл программы. Если нужно задать файл в другом каталоге, то используется соответствующий маршрут (не забывайте только о '\\'

для обозначения ESC-символа в именах маршрутов). Если файла с указанным именем на диске нет, то он создается. Если же файл существует, то он без уведомления удаляется, и заводится новый файл с тем же именем. (Операционная система, поддерживающая версии файлов, создает следующую версию файла.)

Листинг 6.14. Использование динамического массива для считывания набора строк и их записи в файл на диске

```
#include <iostream>
#include <fsteam>                                          // для объектов ifstream, ofstream
using namespace std;

int main (void)
{
  const int LEN = 8; char buf[LEN];                        // короткий буфер для ввода
  int cnt = 0;                                             // счетчик строк
  ofstream f("data.out");                                  // новый файловый объект для вывода
  cout << " Введите данные (или нажмите Enter для завершения): n";
do {                                                       // начало цикла для ввода строк
  int len = 0;                                             // начальная длина данных
  char *data = new char[1]; data[0] = '\0';
  do {                                                     // начало внутреннего цикла для
                                                           // сегментов строк
    cin.get(buf,LEN);                                      // получить следующий сегмент строки
    len +=strlen(buf);                                     // обновить общую длину строки
    char *temp = new char[len+1];
    strcpy(temp,data); strcat(temp,buf);
    delete [] data; data = temp;                           // расширение для длинной строки
    int ch = cin.peek();                                   // что находится слева в буфере?
    if (ch == '\n' || ch == EOF)                           // выход, если новая строка или EOF
      { ch = cin.get(); break; }                           // сначала удалить из ввода
    } while (true);                                        // продолжать до новой строки
  if (len == 0) break;                                     // выход, если вводимая строка пуста
  cout << " строка " << ++cnt << ": " << data << endl;
  f << data << endl;                                       // сохранить данные в файле
  delete [] data;                                          // во избежании "утечек памяти"
  } while (true);                                          // продолжать до пустой строки
  cout << " Данные сохранены в файле data.out" << endl;
  return 0;
}
```

Что если диск переполнен или защищен от записи? Операция создания файла просто не выполняется. Никакой ошибки этапа выполнения не генерируется.

Один из способов решения этой проблемы состоит в вызове компонентной функции `fail()`, возвращающей `true`, если предыдущая операция ввода-вывода была завершена неудачно (по любой причине), и `false` в случае успешного завершения.

```
ofstream f("data.out");          // открывает выходной файл data.out
if (f.fail())                    // проверка на успешное выполнение, отказ, если неудача
  { cout << "Невозможно открыть файл" << endl; return 0; }
```

Многие программисты полагают, что переполнение диска или диск, защищенный от записи,— случаи редкие и данной вероятностью можно пренебречь. С этим трудно согласиться, поскольку такая программа не будет переносимой. В листинге 6.14 мы этим пренебрегли, а напрасно.

После успешного создания объекта ofstream его можно использовать для хранения значений в физическом файле подобно тому, как объект cout используется для вывода значений на экран. Это означает, что при включении операции вставки << битовая последовательность в памяти компьютера преобразуется в последовательность символов, представляющих данные. Для символьных данных преобразование тривиально:

```
f << data << endl;          // записывает массив в выходной файл, а не в cout
```

Как видно, синтаксис доступа к данным здесь тот же, что и для объекта cout. Может ли операция вывода завершиться неудачно? Многие программисты думают, что если файл успешно открыт, нет никакой необходимости проверять каждую операцию. Это не так. Не забывайте, что речь идет о больших объемах данных. Даже современные диски большой емкости могут переполняться, не говоря уже о дискетах и дисках Zip. Поэтому нужно проверять успешное выполнение каждой операции.

```
f << data << endl;                  // сохранение данных в файле на диске
if (f.fail())                       // проверка успешного выполнения операции
  { cout << "Диск переполнен, вывод прекращен" << endl; break; }
```

```
Введите данные (или нажмите Enter для завершения):
First line
 строка 1: First line
Second line
 строка 2: Second line
This is the last line of text
 строка 3: This is the last line of text

   Данные сохранены в файле data.out
```

Рис. 6.17. *Пример выполнения программы из листинга 6.14*

На рис. 6.17 показан пример выполнения программы из листинга 6.14. Для представленных здесь введенных данных файл data.out содержит следующие строки

```
First line
Second line
This is the last line of text
```

Когда файловый объект ofstream выходит из области действия (в листинге 6.14 в конце функции main()), он уничтожается. При этом перестает существовать связь между файловым объектом и физическим файлом, а физический файл закрывается. Исчезновение объекта ofstream не приводит к исчезновению физического файла.

Ввод из файла

Теперь рассмотрим примеры, когда программа использует данные, генерируемые другой программой (например, текстовым редактором) или коммуникационной линией. Для этого можно определить объект класса ifstream (input file stream — поток ввода из файла), представляющий входной файл.

Подобно классу ofstream, класс ifstream определяется в заголовочном файле fstream, который должен включаться в исходный файл программы. Кроме того, аналогично классу ofstream, имя физического файла на диске используется как параметр объекта:

```
ifstream f("amounts.dat");          // открыть файл amounts.dat для ввода
```

Что если заданный файл не найден или его нельзя открыть, поскольку он используется другим приложением? Как и ofstream, объект ifstream все равно создается, но его нельзя будет использовать для ввода. Любая попытка создания объекта ifstream должна сопровождаться проверкой на успешное выполнение:

```
ifstream f("amounts.dat");                  // открыть файл amounts.dat для ввода
if (f.fail())                               // проверка на успешное выполнение
  { cout << "Невозможно открыть файл" << endl;
    return 0; }
```

Если файловый объект ifstream определен успешно, имя объекта ассоциируется с именем физического файла на диске. После этого можно использовать операцию извлечения >> для считывания данных в переменные программы. Вместо объекта cin, представляющего клавиатуру, будет применяться определяемый программистом файловый объект f. Синтаксис доступа к данным в файле тот же, что и для объекта cin. Все другие функции, get(), getline(), setf() и precision(), доступны наряду с манипуляторами и применяются точно так же.

Стоит напомнить, что при использовании операции извлечения данных последовательность считываемых символов преобразуется в битовую последовательность указанного типа (если такое преобразование возможно): int, double, char и пр. Предшествующий пробел или символ новой строки (если он есть) операция пропускает и ищет символы для преобразования, останавливаясь, если находит нечто, что не может быть частью значения (к примеру, символ новой строки). Кроме того, можно считать данные в двоичной форме, а не как последовательность символов. Двоичная форма более компактна, но вид данных в текстовом редакторе или при выводе на экран будет нечитаемым.

Может ли неудачно завершиться операция ввода? Конечно. Более того, при чтении данных из входного файла возможно неуспешное выполнение при достижении конца файла. Для проверки на конец файла можно использовать компонентную функцию eof(), которая возвращает true, если достигается конец файла, и false в противном случае:

```
do {                          // выполнять, пока EOF не даст неуспешное выполнение
  double amount;              // локальная переменная для ввода
  f >> amount;                // получить из файла следующее значение double
  if (f.eof()) break;         // остановить ввод, если больше нет данных
```

Обратите внимание, что предыдущее утверждение несколько туманно. Что значит "достигается конец файла"? Здесь два возможных варианта, нужно понимать разницу между ними. Когда программа читает данные из файла, условие "конец файла" может достигаться немедленно после чтения последней записи в нем. Другой вариант — попытка чтения после завершающей записи в файле.

В языках Ада и Паскаль используется первая интерпретация. В этих языках цикл do, считывающий данные из внешнего файла, выглядит следующим образом:

```
do {                          // структура цикла в языках Ада и Паскаль
  if (f.eof()) break;         // остановить ввод, если больше нет данных
  double amount;              // локальная переменная для ввода
  f >> amount;                // получить из файла следующее значение double
  ... }                       // обработка чтения amount
```

В языках Кобол, C++ и Java применяется вторая интерпретация: условие "конец файла" возникает при попытке программы чтения после конца данных в файле. В этих языках структура цикла do, считывающего данные из внешнего файла, выглядит по-другому:

```
do {                          // структура цикла в языках C++ и Java
  double amount;              // локальная переменная для ввода
  f >> amount;                // получить из файла следующее значение double
  if (f.eof()) break;         // остановить ввод, если больше нет данных
  ... }                       // обработка чтения amount
```

Что произойдет, если сделать ошибку и использовать в программе C++ первую структуру цикла, а не вторую? Последнее значение будет считываться из файла и обрабатываться в остальной части цикла. На следующей итерации eof() вернет false и оператор f >> amount; будет выполнен снова. При отсутствии данных возникнет условие "конец файла", но значение amount в памяти останется

тем же (в большинстве систем). Поскольку программа не предупреждена, что больше данных нет, остальная часть цикла обрабатывает последнее значение (как если бы оно повторилось в строке ввода). На следующей итерации цикл завершится по концу файла.



Осторожно! В C++, когда программа считывает из файла последний элемент, условие конца файла не возникает. Оно возникает при следующем чтении, когда программа пытается считать данные за последним элементом в файле. Избегайте подобной ситуации.

В листинге 6.15 приведена версия программы из листинга 6.13, считывающей данные из файла, а не с клавиатуры. Чтобы легче было сравнивать, здесь закомментированы операторы чтения данных с клавиатуры. Как видно, перейти от чтения данных с клавиатуры к чтению из файла совсем нетрудно. Результаты программы показаны на рис. 6.18.

Листинг 6.15. Использование связанного списка узлов в динамически распределяемой памяти для чтения данных из файла на диске

```
#include <iostream>
#include <iomanip>
#include <fstream>                          // для класса ifstream
using namespace std;

typedef double Item;

struct Node {
Item item;
Node* next; } ;

int main ()
{
  int count = 0;                            // счетчик amount
  Node *data=0, *last;                      // указатели на начало и конец списка
  ifstream f("amounts.dat");                // файл для чтения данных
  if (f.fail())
  { cout << "Невозможно открыть файл" << endl; return 0; }
do {                                        // пока не встретится EOF
  double amount;                            // локальная переменная для ввода
// cout << " Введите сумму (или 0 для завершения): ";
// cin >> amount;                           // получить от пользователя след. значение double
// if (amount == 0) break;
  f >> amount;                              // получить следующее значение double из файла
  if (f.eof()) break;
    Node* q = new Node;                     // создать в динамической области новый узел
    if (q == 0)                             // проверка на успешное выполнение запроса
      { cout << "Нет памяти в динамической области" << endl; break; }
    q->item = amount; q->next = NULL;
    (data == 0 ? data : last->next) = q;
    last = q;                               // last=last->next; также годится
    count++;
    } while (true);
  cout << "\nВсего загружено " << count << " знач.\n";
  if (count == 0) return 0;                 // нет вывода, если нет ввода из файла
  cout << "\Номер Сумма Промежуточный итог\n\n";          // печать заголовка
  cout.setf(ios::fixed);                    // фиксированный формат для double
  cout.precision(2);                        // цифры после десятичной точки
```

```
    double total = 0;                                  // сумма для ввода значений
    int i = 0;
    for (Node *q = data; q != NULL; q = q ->next)      // ОК
      { total += q->item;                              // накопление итоговой суммы
        cout.setw(3); << i+1;                          // номер транзакции
        cout.setw (10); << q->item;                    // значение транзакции
        cout.setw (11); << total << endl;              // текущий итог
      }
    Node *p = data, *r = data;
    while (p != 0)
    { p = p-> next;                                    // предотвращение "зависания" последнего узла
      delete r; r = p; }
    return 0;
}
```

```
Всего загружено 4 значений

Номер    Сумма    Промежуточный итог

  1     330.16          330.16
  2      76.33          406.49
  3      50.00          456.49
  4     120.00          576.49
```

Рис. 6.18. *Результат выполнения программы из листинга 6.15*

Файл amount.dat, который использовался для получения результата для рис. 6.18, содержит следующие строки:

```
330.16
76.33
50.00
120.00
```

Многих программистов функция eof() вполне устраивает, однако она делает программу уязвимой в случае ошибок в форматировании файла ввода.

Предположим, что при наборе числа 50 в третьей строке место 0 была нажата буква 'o'. Когда оператор f >> amount; считает эту строку, он найдет там 5, а затем 'o'. Программа сделает вывод, что вводится значение 5, оставляет символ 'o' в строке ввода и выполняет следующий оператор. На следующей итерации оператор f >> amount находит 'o' в строке ввода, делает вывод, что ввод закончен, и завершается. Выполняется следующий оператор, и программа зацикливается.

Конечно, опечатки такого рода более вероятны при вводе с клавиатуры, чем из файла, ведь файл можно проверить перед выполнением программы. Тем не менее они случаются. Некоторые программисты избегают использования операции >>, так как она слишком уязвима к ошибкам формата ввода. Зацикливание — неприятное явление, как при вводе с клавиатуры, так и из файла. Вместо операции >> они применяют для чтения символьных данных описанные ранее функции get() и getline(). Когда введенная строка находится в памяти, программа может проанализировать данные и сгенерировать интеллектуальное сообщение об ошибке, если они некорректны.

Еще один источник уязвимости — способ завершения файла. В приведенном выше примере символ новой строки вводился после каждого значения, включая последнее — 120. Если за последней записью в файле следует символ новой строки в конце файла, функция извлечения >> при чтении ввода останавливается перед этим символом новой строки. В таком случае условие "конец файла" достигается, только когда программа читает запись, следующую за последней записью.

Что происходит, если последний символ новой строки добавлен не был? Или все значение набраны на одной строке без завершающего символа новой строки? В этом случае функция извлечения данных считывает маркер конца файла и возникает условие "конец файла". Функция eof(), которая вызывается после оператора f >> amount;, возвращает true, и цикл завершается без обработки последнего значения.

Это нехорошо. Программа должна быть написана так, чтобы ее поведение не изменялось в зависимости от того, поместит пользователь (или телекоммуникационное ПО) вслед за последней записью в файле символ новой строки или нет. Чтобы устранить проблему, некоторые программисты избегают применять функцию eof(). Вместо нее они используют старую добрую функцию fail():

```
do {                                  // структура цикла C++ или Java
  double amount;                      // локальная переменная для ввода
  f >> amount;                        // получить из файла следующее значение
  if (f.fail()) break;                // остановить ввод, если больше нет данных
  ..... }                             // остальная часть цикла
```

Функция fail() возвращает значение true, когда операция по какой-то причине (включая достижение конца файла) завершается неудачно. Если вместо 50 набрать 5o, то 5 считывается, а 'o' обнаруживается в потоке ввода при следующей итерации цикла. Оператор f >> amount; ничего не считывает, и функция fail() возвращает true. Цикл ввода завершается. Во-первых, преждевременное завершение лучше, чем зацикливание. Во-вторых, после завершения цикла программа может проанализировать ситуацию и сгенерировать сообщение об ошибке в случае, если это произошло преждевременно.

Во втором примере, когда за значением 120 не следует символ новой строки, возникает условие "конец файла", но функция fail() возвращает false, так как значение 120 было считано оператором f >> amount; корректно. Лишь на следующем проходе цикла, когда программа пытается прочитать следующее значение, эта функция возвращает true. Следовательно, последнее значение в файле обработано корректно.

Файловые объекты ввода и вывода

Кроме ifstream и ofstream библиотека iostream в C++ определяет большое число других классов. В 99% случаев программисту не потребуется ничего знать о них. Здесь упоминается только один потоковый класс fstream, так как он объединяет в себя характеристики классов ifstream и ofstream.

При создании объектов типа ifstream и ofstream режим открытия не указывается: ifstream создается по умолчанию для чтения, а ofstream — для записи. Для объектов класса fstream задается режим открытия. Для этого при создании объекта используется второй аргумент:

```
fstream of("data.out,ios::out);              // выходной файл
fstream inf("amounts.dat,ios::in);           // входной файл
```

Режим ввода задается по умолчанию, его можно опустить. В число других доступных режимов входят ios::app (файл открывается для добавления данных в конец), ios::binary (файл открывается в двоичном, а не в текстовом формате) и другие. Эти режимы реализованы как двоичные флаги. Если нужно, их можно комбинировать с помощью двоичной операции "включающее ИЛИ" — ('|').

```
fstream mystream("archive.dat",ios::in|ios::out);        // ввод-вывод
```

В C++ существует несколько способов проверки успешного или неуспешного выполнения операции с файлом. Кроме описанной выше функции fail(), можно применять функцию good():

```
fstream inf("amounts.dat",ios::in);                      // входной файл
if (!inf.good())                                         // еще один способ
  { cout << "Невозможно открыть файл" << endl;
    return 0; }
```

Можно даже интерпретировать файловый объект как числовое значение. Когда операция завершается неудачно, оно равно 0, а в случае успешного выполнения содержит нечто, отличное от нуля. Вот еще один пример проверки успешного открытия файла:

```
fstream inf("amounts.dat,ios::in);                    // входной файл
if (!inf)                                             // еще один способ
  { count << "Невозможно открыть файл" << endl; return 0; }
```

Тот же синтаксис можно применять для проверки успешного выполнения операций чтения и записи. Например, нетрудно подсчитать число символов в файле с помощью функции `get()` с односимвольным параметром. Если операция чтения завершается неудачно (из-за достижения конца файла или по любой другой причине), функция возвращает 0 и это значение можно использовать для завершения цикла `while`.

```
int count = 0; char ch;
while (inf.get(ch))              // остановка, когда с объектом что-то не так
  count++;
cout << "Всего символов: " << count << endl;
```

Обычно закрывать файлы не требуется. Они закрываются, когда в конце области действия уничтожается ассоциированный с файлом объект. Однако иногда возникает необходимость закрыть файл явно. Это делается с помощью функции `close()`:

```
inf.close();                    // закрыть файл
```

Такая необходимость может возникать, если требуется закрыть файл до того, как завершится область действия его файлового объекта, например когда открывается несколько файлов и следующий файл открыть не удается. В такой ситуации до попытки восстановления или завершения программы лучше явно закрыть все открытые файлы. Может понадобиться закрывать файл и в том случае, если нежелательно хранить несколько файлов открытыми, например когда данные считываются из одного файла, обрабатываются в памяти, а затем результаты записываются в другой файл, чтобы их можно было использовать позднее.

В листинге 6.16 показана модифицированная программа из листинга 6.15. Кроме вывода результатов на экран, она сохраняет отчет в файле `amounts.rep`. Здесь путем сравнения файловых объектов с нулем проверяется успешность выполнения операций ввода-вывода. В C++ это общепринятый подход. Входной файл закрывается в конце ввода.

Листинг 6.16. Чтение из файла, вывод на экран и в выходной файл

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

typedef double Item;
struct Node {
  Item item;
  Node* next; } ;

int main ()
{
  int count = 0;                                    // счетчик amount
  Node *data=0, *last;                              // указатели на начало и конец списка
  fstream inf("amounts.dat",ios::in);               // файл для чтения данных
  if (!inf) { cout << "Невозможно открыть файл" << endl; return 0; }
```

```
do {                                          // пока не встретится EOF
   double amount;                             // локальная переменная для ввода
   inf >> amount;                             // получить следующее значение double из файла
   if (!inf) break;
     Node* q = new Node;                      // создать в динамической области новый узел
     if (q == 0) { cout << "Нет памяти в динам. области" << endl; break; }
     q->item = amount; q->next = NULL;        // заполнить данными
     (data == 0 ? data : last->next) = q;
     last = q; count++;
     } while (true);
   inf.close();                                          // файл больше не нужен
   fstream of("amounts.rep.ios::out);                    // файл для записи данных
   if (!of) { cout << "Невозможно открыть выходной файл" << endl; }
   cout << "\nВсего загружено " << count << " знач.\n";
   of << "\nВсего загружено " << count << " знач.\n";
   if (count == 0) return 0;                             // нет вывода, если нет ввода из файла
   cout << "\Номер Сумма Промежуточный итог\n\n";         // печать заголовка
   of << "\Номер Сумма Промежуточный итог\n\n";          // печать заголовка
   cout.setf(ios::fixed); cout precision (2);            // точность для экрана
   of.setf(ios::fixed); of.precision(2);                 // точность для файла
   double total = 0; int i = 0;                          // промежуточный итог, счетчик строк
   for (Node *q = data; q != NULL; q = q ->next)         // ОК
     { total += q->item;                                 // накопление итоговой суммы
       cout << setw(3) << i;                             // номер транзакции
       cout << setw(10) << q->item;                      // значение транзакции
       cout << setw(11) << total << endl;                // текущий итог
       of << setw(3) << ++i << setw(10) << q->item;      // транзакция
       of << setw(11) << total << endl; }                // текущий итог
   Node *p = data, *r = data;
   while (p != 0)
   { p = p-> next;                                       // предотвращение "зависания" последнего узла
     delete r; r = p; }
   return 0;
}
```

Вы видите, операторы форматирования данных при выводе в файл те же, что и при выводе данных на экран. При вводе данных из рис. 6.18 программа из листинга 6.16 создает выходной файл со следующими данными:

Всего загружено 4 значения

Номер	Сумма	Промежуточный итог
1	330.16	330.16
2	76.33	406.49
3	50.00	456.49
4	120.00	576.49

Это все, что нужно знать о работе с файлами при помощи библиотеки iostream. Данная библиотека содержит значительно больше средств, чем здесь описано, но пока что не стоит вдаваться в детали. Сначала нужно познакомиться с классами и наследованием. На самом деле даже после изучения классов и наследования может не потребоваться знать больше того, о чем рассказывалось выше. Библиотека iostream предлагает несколько способов выполнения одних и тех же действий, и вовсе не обязательно изучать все их сразу. Вместо этого лучше обратить

внимание на базовые средства языка и убедиться в понимании основных концепций. Тем самым можно подготовиться к использованию других библиотечных средств или их пониманию, если они встречаются в программах других разработчиков.

Итоги

В данной главе изложен достаточно сложный материал. Мы рассмотрели несколько примеров использования указателей в C++. Первый предусматривал применение указателей для ссылки на обычные переменные, распределяемые в стеке, и реализации альтернативного метода доступа к данным переменным через псевдонимы. За исключением передачи параметров через указатель (о чем подробнее рассказано в следующих главах) этот метод не особенно полезен. Но, некоторые программисты считают, что подобная техника делает их программы более понятными. Поэтому нужно быть готовым к тому, чтобы иметь дело с такого рода операциями с указателями в унаследованном программном коде.

Второй тип использования указателей состоит в выделении памяти для отдельных переменных не в стеке, а в динамически распределяемой области. Переменные в динамически распределяемой области памяти не имеют имен, и указатели — единственный способ доступа к их содержимому. В то же время применение переменных в динамически распределяемой области вместо обычных переменных в стеке не дает практических преимуществ, его следует избегать. Некоторые полагают, что такая техника дает возможность снизить вероятность переполнения стека, а потому нужно быть готовым к применению подобных операций с указателями в унаследованных программах.

Два других вида использования указателей вполне законны, полезны и весьма распространены в программах на C++. Неименованные динамические массивы представляют превосходную альтернативу именованным массивам C++, размер которых определяется на этапе компиляции. Динамические массивы исключают опасность переполнения массива и непроизводительной траты памяти. Операции с ними выполняются быстро и достаточно просто. При определении динамических массивов убедитесь в том, что массив не распределяется и не освобождается повторно. Место для определения такого массива нужно выбирать так, чтобы это не сказалось негативно на производительности программы.

Еще одним ценным методом применения указателей являются связанныеструктуры. Это самый гибкий способ распределения памяти, поскольку память не резервируется заранее, а выделяется по мере необходимости. Но операции с указателями при включении и удалении узлов достаточно сложны, их нельзя назвать интуитивно понятными. Это же относится к операциям перебора списка. Ошибки в операциях с указателями очень трудно обнаружить. Они не всегда проявляются в некорректном поведении программы. Распределение и освобождение памяти выполняется фрагментарно, индивидуально для каждого узла, что может сказаться на производительности программы.

Если возникает необходимость использовать связанные списки, нужно рассмотреть альтернативные варианты. Один из них — динамические массивы, второй — применение стандартной библиотеки шаблонов с готовыми вариантами реализации таких структур, как связанные списки, очереди, деревья и пр. Подобные библиотеки позволяют программисту комбинировать гибкость динамического распределения памяти с простотой применения, так что нужно этим пользоваться.

В последних разделах главы рассмотрены последовательности данных, длина которых не определена заранее. Это физические файлы. Здесь демонстрировались способы определения библиотечных объектов, позволяющих использовать при работе с файлами такие же операции, что и при вводе с клавиатуры и выводе на экран. Применение файлов позволяет программе сохранять данные на диске,

где они могут находиться неопределенно долго. После сохранения на данные не повлияет аварийное или нормальное завершение программы или сбой питания. Кроме того, что еще более важно, данные могут использоваться другой программой в другое время и/или в другом месте. Это существенно расширяет гибкость информационных систем.

Данная глава завершает обсуждение необъектно-ориентированных средств C++. В следующих главах мы начнем детальное обсуждение функций и классов C++, изучим создание объектно-ориентированных программ. Весьма впечатляющая тема! Как уже упоминалось, объектно-ориентированный подход является, вероятно, единственным, помогающим разработчикам создавать программы из относительно независимых компонентов и выражать свои идеи, понятные сопровождающему приложение программисту, непосредственно в исходном коде. Такие навыки не вырабатываются автоматически, просто в процессе изучения языка. Нужно надеяться, что дальнейшее чтение книги поможет вам освоить это важное искусство.

Часть II

Объектно-ориентированное программирование на C++

В этой части книги описываются базовые инструментальные средства объектно-ориентированного программирования на C++. Объектно-ориентированное программирование — это в первую очередь использование функций, ведь каждая операция с объектом должна реализовываться через вызов функции. Функции C++ — достаточно сложная тема, и в главе 7 рассказывается об их синтаксисе. Передача параметров C++ может представлять трудности. Остается надеяться, что данная глава поможет читателям овладеть этими важными навыками.

В главе 8 продолжается обсуждение функций и поясняется, как их использовать. Она знакомит с критериями сцепления, связности, инкапсуляции и сокрытия информации. В ней рассказано, как писать легко читаемые и независимые функции, продемонстрировано, что основные преимущества объектно-ориентированного программирования могут достигаться и без использования объектов C++ — за счет проектирования функций, вызываемых клиентскими программами (вместо прямого доступа к полям структуры). Показываются и ограничения объектно-ориентированного программирования с применением функций, перечисляются цели использования классов C++. Эта глава очень важна для выработки верной интуиции при объектно-ориентированном программировании.

В главе 9 читатель познакомится с вершиной программирования на C++ — классами. Здесь описывается синтаксис определения классов C++, обсуждаются функции-члены и элементы данных, управление доступом к компонентам класса, инициализация, уничтожение объектов и другие технические детали работы с объектами. Читатели найдут немало сложных подробностей, но без этого не обойтись. Классы C++ — тема не простая. Пусть эти детали не скрывают от вас основной цели применения классов — доведение до сопровождающего приложение программиста общего смысла обработки данных и передачи информации между функциями.

Глава 10 посвящена операторным функциям — примечательной части синтаксиса C++. Операторные функции введены в язык для поддержки принципа, согласно которому программа должна иметь возможность делать с объектами класса все, что она может делать с обычными числовыми значениями, включая сложение, вычитание и т. д. Такая концепция не очень важна с точки зрения разработки ПО, но позволяет дополнить код C++ превосходными синтаксическими конструкциями.

В главе 11 обсуждаются опасности непродуманного использования конструкторов и деструкторов C++, поясняется, как можно вовремя увидеть данную опасность. В ней приводятся некоторые методы, позволяющие избежать порчи содержимого и "утечек" памяти. Это очень важно, так как неопытный программист может нанести немало вреда, некорректно работая с инициализацией объектов C++.

Глава 7

Программирование с использованием функций C++

Темы данной главы

- ✔ Функции C++ как инструмент разбиения программы на модули
- ✔ Передача и преобразование аргументов
- ✔ Передача параметров в C++
- ✔ Встраиваемые функции
- ✔ Параметры со значениями по умолчанию
- ✔ Перегрузка имен функций
- ✔ Итоги

В предыдущих главах мы рассмотрели основные концепции языка C++, позволяющие реализовать любые сложные требования, стоящие перед компьютерной системой.

Встроенные типы данных C++ дают программисту возможность привести объекты в соответствие с решаемой задачей. Они предоставляют богатый выбор для числовых диапазонов и точности. С помощью операций C++ вводимые значения комбинируются в гибкие и мощные выражения для вычисления необходимого результата, а управляющие структуры помогают организовать требуемую последовательность вычислений и изменить ход вычислений в зависимости от условий, либо итеративно повторять их.

Мы рассмотрели также средства C++, обеспечивающие агрегирование компонентов, и типы данных, определяемые программистом. С их помощью программист может комбинировать отдельные логически связанные значения данных для работы с ними как с целыми блоками. Разработчику они позволяют выражать свои идеи в понятном сопровождающему приложение программисту виде, показывая, что эти компоненты должны быть связаны. Массивы дают программисту возможность комбинировать элементы, подлежащие одинаковой обработке в программе. Наконец, мы обсудили управление динамической памятью и файлами. Они расширяют мощность и гибкость обычных массивов и позволяют преодолеть свойственные массивам ограничения.

Далее нам предстоит углубиться в еще один инструмент модульности и агрегирования C++. Комбинируя отдельные элементы в функции, программист может интерпретировать их как единый логический блок. Разбиение программы на отдельные функции — мощный инструмент разделения труда. Разные программисты могут создавать отдельные функции параллельно.

В данной главе мы изучим методы написания функций C++. Основное внимание будет уделено коммуникациям с функциями — тому, как они обмениваются данными, методам передачи параметров и возврата значений из функций. Эти методы зависят от того, модифицирует ли функция свои аргументы или сохраняет те значения, которые они имели при вызове. Кроме того, они зависят от типа параметров: это может быть встроенный тип C++, массив, определенная программистом структура или класс.

Часть главы посвящена методам проектирования функций, позволяющим ослабить ограничения, связанные с именами функций, включая значения параметров по умолчанию и перегрузку имен функций (overloading). Эти методы существенно расширяют возможности выбора программистом различных вариантов реализации. Кроме того, читатели узнают, как преодолеть отрицательное влияние на производительность при использовании встраиваемых функций (inline) и что происходит, когда подставляемые в вызове функции аргументы не соответствуют по типу формальным параметрам, определенным в заголовке функции.

Весьма амбициозная программа. Функции C++ — гибкий и мощный механизм. Они предоставляют программисту изумительные возможности выбора разных вариантов их реализации. Попробуем извлечь из этого некоторую выгоду.

Весь материал данной главы крайне важен для освоения классов C++. Прочитайте его, экспериментируйте с примерами, и их использование окажется не таким уж сложным.

Функции C++ как средства разбиения программы на модули

Как и в других языках, в C++ программист скрывает сложность компьютерных алгоритмов в относительно небольших модульных единицах — *функциях*. Каждая функция представляет собой набор операторов языка, предназначенных для достижения определенной цели. Эти операторы могут представлять собой простое присваивание, сложные управляющие конструкции или вызовы других функций — стандартных библиотечных функций, созданных в предыдущих проектах или специально разработанных для конкретного проекта. С точки зрения программиста, разница между реализациями различных функций в том, что код специально разработанных для проекта функций можно проверить. Программист, использующий библиотечные функции как серверы для собственной функции, не знает способы их реализации. Ему известно лишь описание интерфейса серверной функции: какие параметры нужно ей передавать, что именно вычисляет функция, какие результаты получаются из значений на входе, какие ограничения и исключения применяются к данной функции.

Это не означает, что исходный код библиотечных функций является торговым секретом. Иногда (но не часто) он свободно распространяется. На самом деле ограничение знаний программиста интерфейсом функции имеет свои преимущества: это уменьшает сложность программы. Изучение исходного кода функций оправдано только в том случае, если функция содержит ошибки, которые можно исправить. Это имеет место в функциях, определяемых программистом и создаваемых для конкретного проекта. Но даже в этих функциях задача анализа взаимодействия между функциями должна ограничиваться изучением интерфейсов функций, а не их реализации.

Именно с этой точки зрения мы будем рассматривать различные методы коммуникаций между функциями. Эти методы позволяют сопровождающему приложение программисту или другому разработчику изучить только интерфейсы функций и не читать их исходный код. Такой метод предпочтительнее метода, требующего и анализа исходного кода функций.

Вызывающие (клиентские) функции интерпретируют вызываемую (серверную) функцию как единый блок. В вызове функции задается ее имя и фактические аргументы (если они есть). Вызывающая программа не знает, *как* именно функция (сервер) выполняет свою задачу. Она знает только, *что* делает серверная функция и каковы спецификации интерфейса. Следовательно, использование вызовов функций упрощает клиентский код, который достигает собственных целей без детальных шагов, абстрагируя их в виде вызова функции.

Функция представляет собой небольшую единицу модульности. Применение функций позволяет разработчику организовать большую программу в виде более мелких и лучше управляемых блоков. Разные функции можно распределить между программистами, ускорив тем самым разработку большого приложения.

Если один и тот же алгоритм нужно использовать в нескольких местах программы, то его реализация как функции позволяет разработчикам вызывать алгоритм в разных местах программы, а не воспроизводить детально в клиентском коде. В результате уменьшается объем объектного кода и упрощается повторное использование фрагментов программы. При сопровождении программы небольшие функции проще понять, с ними легче работать, чем с большой монолитной программой.

Функции, применяемые как модульные единицы для организации программного кода, можно поместить в библиотеку. В этом случае их можно включать в другие программы и вызывать из них, что также увеличивает повторное использование кода.

Грамотное проектирование функции очень важно для ее читабельности, независимости разных частей программы и уменьшения сложности приложения. При этом очень большое значение имеет хорошее функциональное проектирование кода. При использовании функций программисту нужно координировать:

- Объявление функции (ее прототип), включая имя функции, возвращаемый ею тип и типы ее параметров
- Определение функции — ее заголовок и реализацию тела функции
- Вызов функции, т. е. ее имя и имена (или значения) фактических аргументов

Все эти элементы должны быть скоординированы. На первый взгляд, это не очень сложно, поскольку элементов только три. Действительно, большинство программистов, как правило, делают все верно. Проблема возникает в том случае, когда программист в каком-то из этих трех пунктов ошибается. Хотя такие ошибки не очень часты, они приводят к весьма серьезным проблемам.

Объявление функции

C++ требует, чтобы до обработки вызова функции компилятор видел ее объявление или определение. Следовательно, в исходном файле, где вызывается функция, она должна перед вызовом объявляться или определяться. Именно поэтому в программировании на C++ важным компонентом является применение необходимых прототипов функций.

В объявлении функции типы параметров и возвращаемое функцией значение (если оно имеется) должны описываться наряду с именем функции. Если функция вызывается в нескольких файлах, то она должна объявляться в каждом из них:

```
Возвращаемый_тип имя_функции(тип1 парам1, тип2 парам2, ...);
```

Если функция не возвращает значения, то возвращаемый ею тип не просто опускается, а описывается как void, однако, если возвращаемый тип опущен, это не синтаксическая ошибка. Компилятор предполагает, что программист хочет использовать вместо void тип int, поэтому пропуск типа вполне допустим. Пропуск возвращаемого функцией типа — популярный метод программирования на C++. Однако он затрудняет понимание программы, и сопровождающему ее программисту приходится тратить время на уяснение сути вещей. Если функция возвращает тип int, нужно так и сказать об этом. Пропуск типа — порочная практика. Кроме того, некоторые компиляторы могут выводить предупреждение, что данный стиль определения функции устарел.

```
add(int x, int y);                          // возвращает int: плохой стиль
void PutValues(int val, int cnt);           // не возвращает значения: тип void
```

Функция может возвращать только одно значение. Если клиентской программе нужно получать от нее несколько значений, то функция может возвращать структурную переменную, хотя это замедляет выполнение программы. Кроме того, функция может модифицировать любое число глобальных переменных, определенных в файле вне этой функции. Как будет показано далее, ни один из этих методов нельзя считать хорошей практикой программирования — они ведут к ошибкам. Функция может также модифицировать значения своих аргументов. Все эти методы достаточно сложны, но не отчаивайтесь. Вы их освоите.

Определения функций

В определении функции реализуется ее алгоритм на C++. Определение начинается со строки заголовка, где указывается тип возвращаемого значения, имя функции (определенное программистом идентификатора). Оно содержит также список разделенных запятыми параметров (имена и типы). Разница между заголовком функции и прототипом функции в том, что прототип заканчивается точкой с запятой, а заголовок — нет. Еще одно отличие в параметрах: в прототипах они обязательны, а в заголовках функции — нет. На самом деле, если параметр в теле функции не используется, его имя в заголовке также необязательно, но вряд ли кому в голову придет писать такую функцию.

Тело функции — это блок со своей областью действия. Как и в любом коде C++, если не используется управляющая конструкция или вызов другой функции, операторы в теле функции выполняются последовательно.

```
void PutValues(int val, Int cnt)
{ cout << "Значение " << val << " встречается ";
  cout << cnt << " раз" << endl;
  return; }                              // не обязательно, функция void
                                         // не возвращает значения

int add (int x, int y)
{ count++;                               // глобальная переменная модифицирована
  return x+y; }                          // оператор return и возвращаемое
                                         // значение необязательны
```

Для функции void оператор return необязателен. Такие операторы можно использовать в любом месте функции void, но они не возвращают значения. Выполнение оператора return завершает выполнение функции и возвращает управление в вызывающую программу. Отличная от void функция должна содержать по крайней мере один оператор return, а может использоваться несколько операторов return. Каждый return в этом случае должен возвращать значение, тип которого указан в заголовке функции (или преобразовываться к данному типу в операторе return).

Вызовы функций

В языках Паскаль, Ада и др. различаются процедуры и функции. Процедуры там не возвращают значений, но могут изменять свои аргументы и глобальные переменные. В клиентском коде они вызываются только в отдельных операторах, но не в выражениях. Функции в этих языках возвращают значения, но не могут влиять на аргументы. В клиентском коде их нельзя использовать как отдельные операторы, а нужно включать в выражения (или как r-значение в операторе присваивания):

```
a = add(b,c) * 2;          // использование возвращаемого значения в выражении
PutValues(a,5);            // вызов функции в операторе
b = PutValues(a,5)*2;      // нонсенс: нет возвращаемого значения
```

В C++ процедуры и функции не различаются. Функции C++ могут как возвращать значения, так и иметь побочные эффекты. Тип возвращаемого значения — это любой встроенный или определяемый программистом тип, но массивы быть возвращаемым значением не могут.

Если функция имеет тип `void`, она работает как процедура и не возвращает значения в вызывающую программу. Ее нельзя использовать в выражении. Такая функция должна вызываться в отдельном операторе.

В отличие от языков Паскаль и Ада, C++ позволяет клиенту игнорировать возвращаемое функцией значение и использовать вызов функции как вызов процедуры. Это означает, что отличная от `void` функция может быть частью выражения и вызываться в отдельном операторе. Когда вызывающая программа использует вызов функции как оператор, единственная цель такого вызова — побочные эффекты (изменение глобальных переменных):

```
add(b,c);              // корректный синтаксис, даже если в нем нет смысла
```

Это неправильная практика программирования. Если функция возвращает значение, его следует употребить в клиентском коде. Между тем, в C++ существует немало библиотечных функций, возвращающих отличные от `void` значения, которые используются редко, например `sctcpy()` и `strcat()`.

Тело функции в фигурных скобках определяет действие, выполняемое при вызове функции. При этом говорят, что операция вызова `()` применяется к имени функции и разделенным запятыми аргументам вызова:

```
PutValues(17,14);                  // применяется оператор вызова функции
```

Большинство не рассматривают вызов функции в терминах применения операции вызова. Достаточно думать о списке аргументов в скобках. Между тем в "продвинутом" программировании на C++ важно не забывать о том, что вызов функции C++ представляет собой применение операции вызова. Более того, эту операцию можно использовать в других контекстах, придавая ей иной смысл.

Если определение функции-сервера лексически предшествует определению функции-клиента, то компилятор перед обработкой вызова функции видит определение сервера. В этих случаях определение сервера может служить также его объявлением, но многие программисты не полагаются на лексический порядок функций и применяют прототипы — это становится привычкой.

В программе функция может определяться только один раз, а ее прототипы — повторяться столько раз, сколько необходимо (и даже более). Часто прототипы функции размещают в отдельных заголовочных файлах в каталоге проекта. Такие файлы включаются в программы, вызывающие эти функции. Часто программисты включают в исходный файл не только вызываемые функции, но и другие, пока не используемые в программе. Это проще, чем изучать, кто и что вызывает и в каком файле. Компилятор воспринимает такие действия нормально — лишние прототипы он просто игнорирует. В то же время заголовочные файлы не должны

игнорироваться программистами, занимающимися сопровождением программы. Неразборчивое использование прототипов затрудняет понимание зависимости между различными частями программы.

C++ позволяет опускать в прототипах функций имена параметров. Они нужны только в определении функции. Многие программисты опускают имена параметров, поскольку компилятор в них не нуждается.

```
void PutValues(int, int);              // а что с параметрами?
```

Такое допустимо, когда типы параметров разные, роли параметров вполне понятны программистам, занимающимся сопровождением ПО (например, в часто используемой библиотечной функции), а прототипы скрыты в заголовочном файле. Для функции, определяемой программистом, имена параметров поясняют, для чего именно они используются.

Некоторые программисты объявляют прототипы функций в клиентском коде не в начале файла, а непосредственно внутри делающей вызов клиентской функции (в целях документирования). Тем самым явно сообщается, что именно эта функция использует серверную функцию (в отличие от многих других функций в том же файле пусть вас не вводит в заблуждение простота приводимых для иллюстрации примеров, на самом деле для разработки ПО вопрос вполне серьезный):

```
void Client(void)
{ void PutValues(int value, int count);              // список зависимостей
  int val, cnt;
  cout << "Введите значение и его номер: ";
  cin >> val >> cnt;
  PutValues(val, cnt); }
```

Если функция не имеет параметров, в ее прототипе и определении могут использоваться пустые круглые скобки или скобки с ключевым словом void.

```
int foo(); int f(void);          // функции без параметров
```

Однако в вызове функции для указания оператора вызова функции используются только пустые скобки.

```
foo(); f();              // круглые скобки допускаются и обязательны
```

Почему ключевое слово void разрешается включать в определение и объявление функции, но не в ее вызов? Чтобы облегчить жизнь разработчикам компиляторов. Если бы void допускалось в вызове функции, компилятор мог бы решить, что это прототип функции, включенный в клиентский код:

```
f(void);                 // это не вызов функции, а прототип
```

Но где же тип возвращаемого значения? Как компилятор поймет, что это прототип? Очень просто: он подумает, что программист просто опустил int. Такая практика программирования не приветствуется, но допускается. Кстати, это обещанный ответ на вопрос, заданный в главе 2.

Преобразование типов аргументов

Поскольку C++ — язык с сильным контролем типов, в вызовах функции C++ для всех формальных параметров должны использоваться корректные типы и число фактических аргументов. В теле функции фактические аргументы применяются как значения соответствующих формальных параметров. Если число и порядок аргументов не отвечают порядку или числу формальных параметров, это однозначно считается синтаксической ошибкой:

```
PutValues(25);           // ошибка - пропущен один аргумент
```

Если число и порядок аргументов корректны, а их типы не совпадают с типами параметров, при проверке соответствия параметров и аргументов также возникнет синтаксическая ошибка. Типы считаются несовместимыми, если преобразование не имеет смысла. Например, если один из типов определяется программистом (структура или класс), а другой — простой встроенный тип, массив или другой определяемый программистом тип, одно значение не может использоваться вместо другого.

Предположим, например, что a1 — определенный программистом тип Account (структура), а a2 — массив (не важно, какого типа). Тогда в следующем вызове функции будет две синтаксических ошибки:

```
PutValues(a1,a2);              // несовместимые типы - две ошибки
```

Причина в том, что в своем теле эта функция имеет дело с двумя целочисленными параметрами. Операции, допустимые с такими параметрами, не допускаются с объектами Account или с массивами. С ними нельзя делать то, что можно делать с числом (складывать, умножать, сравнивать и пр.). Так можно работать с их отдельными компонентами, но это уже совсем другая история.

Рассмотрим теперь функцию, рисующую квадрат по некоему параметру определенного программистом типа Square:

```
void draw(Square);
```

Не важно, каков именно состав или свойства типа Square. Такой клиентский код некорректен:

```
draw(5);                       // несовместимые типы - синтаксическая ошибка
```

Это также понятно. С числом нельзя делать то, что можно делать со структурой (например, обращаться к компоненту с помощью операции точки). Подобно другим современным языкам программирования, в C++ здесь используется жесткий и бескомпромиссный подход.

В случае менее несовместимых объявленного и фактического типов возможно преобразование. Несоответствие означает, что типы разные, но они могут иметь общие операции, а следовательно, допускается использование значений одного типа вместо значений другого. Такие типы рассматриваются как совместимые.

Преобразование из "меньших" в "большие" типы выполняется для некоторых типов неявно перед вычислениями. Аргументы типа enum преобразуются в int, аргументы типа char, unsigned char и short — также в int. Аналогично, в int (или unsigned int на машине, где int не больше, чем short) преобразуются аргументы типа unsigned short. Аргументы типа float приводятся к типу double. Такое преобразование аргументов "безопасно" — нет угрозы потери точности или применения операции, не определенной для "меньшего" типа.

Если после преобразования тип аргумента не соответствует типу формального параметра или аргумент не подлежит преобразованию к "большему" типу (имеет тип int, long или double), то используется неявное преобразование: любой числовой тип (включая unsigned) может преобразовываться в любой другой числовой тип. Это делается даже в том случае, если преобразование потенциально может привести к потере точности (например, преобразование из double в int). Нулевой фактический аргумент можно преобразовать в формальный параметр любого числового типа или к типу указателя, даже если возможна потеря точности.

Рассмотрим, например, снова функцию PutValues(). Что происходит, если передаются аргументы double, а не int? Они без всяких предупреждений приводятся к типу int. Впрочем, некоторые компиляторы C++ выводят предупреждающее сообщение. Тем не менее в C++ это вполне законно:

```
double x = 20, y = 5;          // целые преобразуются в double
PutValues(x,y);                // double преобразуются в int
```

Как сообщить компилятору, что несоответствие типов — нормальная ситуация и вы знаете, что делаете? Обычно для этого применяется явное приведение типа, преобразующее значение одного типа в значение другого.

```
PutValues((int)x,(int)y);          // явное приведение - для компилятора
                                   // и сопровождающего программиста
```

Вот еще один способ — приведение типа в стиле функции:

```
PutValues(int(x),int(y));          // альтернативный синтаксис
                                   // для явного приведения типа
```

Обратите внимание, что явное приведение типа позволяет разработчику сообщить о своих намерениях не только компилятору, но и сопровождающему приложение программисту, так что ему не нужно будет гадать, что происходит.

Те же правила преобразования применяются в случае, если не совпадают объявленный тип возвращаемого значения и его фактический тип. Если возвращаемый тип "меньше" объявленного, то фактическое значение преобразуется к объявленному типу. Если тип фактического возвращаемого значения не меньше или преобразование к "большему" типу выполнить невозможно (фактическое значение имеет тип `int`, `long` или `double`), то возвращаемое значение преобразуется к объявленному типу.

Эти преобразования аргументов аналогичны преобразованиям типов в выражениях C++. Цель состоит в том, чтобы сделать операции по возможности "законными". В этом плане C++ "мягче", чем другие современные языки. Более того, применение наследования, конструкторов и перегруженных операций преобразования (о которых рассказывается далее) делает C++ еще менее строгим в отношении преобразования типов аргументов. Это замечательно, если такие преобразования действительно соответствуют тому, что задумывалось разработчиком. Хуже, если разработчик делает ошибку, и компилятор сообщает ему об этом. Но в любом случае, применение неявного преобразования типов существенно осложняет жизнь программисту, сопровождающему программу.

Лучше всего, когда типы аргументов и возвращаемых значений в точности совпадают, или используется явное приведение типов, помогающее программисту понять, что происходит.

Передача параметров в C++

В C++ предусмотрено три режима передачи параметров: по значению, по указателю и по ссылке.

Когда параметр передается по значению, изменения этого параметра в функции не влияют на значение фактического аргумента, использованного при вызове функции. Если параметр передается по указателю или по ссылке, изменения в параметре влияют на фактические аргументы клиента. Кроме того, имеется специальный режим для передачи параметров-массивов.

Мы рассмотрим разные режимы передачи параметров, их синтаксис и семантику, а также попробуем сформулировать рекомендации по использованию режимов передачи параметров в C++, позволяющие добиться наилучшей производительности и сообщить о намерениях разработчика сопровождающему приложение программисту.

Передача по значению

При вызове функции значения аргументов могут передаваться как переменные (или символьные константы), выражения или литеральные значения соответствующих типов:

```
int n = 22, cnt = 20;
PutValues(n,cnt);                  // аргументы как переменные
PutValues(2*n,cnt-11);             // аргументы как выражения
PutValues(18,14);                  // аргументы как литеральные значения
```

При выполнении параметры функции интерпретируются как локальные переменные, область действия которых — тело функции. Имя параметра известно компилятору и в области действия функции (между открывающей и закрывающей фигурной скобкой) ссылается на конкретный адрес памяти. Вне области действия функции это имя неизвестно. Даже если вне функции данное имя используется для какой-то другой цели, оно никогда не ссылается на ту же область памяти, что и параметр функции.

Параметры определяются (распределяются и инициализируются) при вызове функции. Память для параметров отводится в стеке программы и инициализируется значением фактических аргументов. Формальные параметры представляют собой отдельные копии значений фактических аргументов. При завершении функции (выполнении оператора `return` или достижении закрывающей фигурной скобки) эти копии уничтожаются. Рассмотрим, например, следующую примитивную функцию, возвращающую сумму своих аргументов:

```
int add (int x, int y)             // x, y создаются и инициализируются
{ return x+y; }                    // y и x уничтожаются
```

Поскольку параметры (копии фактических аргументов) находятся в области действия вызывающей функции, они могут модифицироваться в теле функции.

```
int add (int x, int y)
{ x = x+y;                    // допустимо - изменяется x
  return x;                   // новое значение копируется в переменную клиента
 }                            // модифицированная копия аргумента уничтожается
```

Фактические аргументы не находятся в области действия вызываемой функции. Значения передаются только в одном направлении — от вызывающей функции вызываемой. Если вызванная функция модифицирует параметры, то изменения не передаются обратно в область вызывающей функции после уничтожения параметров. Рассмотрим следующий код клиента:

```
int a= 2, b = 3, c; ...
c = add(a,b);          // переменная 'a' не изменяется в области действия клиента
```

Вызов по значению — "естественный" режим передачи параметров в C++. В этом режиме значения фактических аргументов (переменные, выражения, значения литералов) копируются во временные переменные, представляющие параметры функции. После этого переменные в области действия клиента не связаны с данными копиями. Функция может манипулировать ими, а при ее завершении они уничтожаются. Изменения в копиях внутри функции никак не влияют на значения аргументов в клиенте.

Все это понятно. Когда параметры передаются по значению, фактическими аргументами могут быть любые значения, например выражения или литералы. Сами эти r-значения не могут и не должны изменяться в функции. Например, в следующем вызове гарантируется, что первый аргумент в вызове функции не изменяется:

```
c = add(2*5,b);            // передача как r-значения 2*5 функции
```

Для побочных эффектов C++ предусматривает передачу параметров по указателю и по ссылке. Такие режимы более сложны, чем передача параметров по значению. Некорректное их использование затрудняет сопровождение программы, а корректное — способствует ее высокой производительности и читабельности.

Вызов по указателю

Переменная-указатель содержит адрес другой переменной (поэтому она и называется указателем — указатели ссылаются на другие объекты программы). Переменными программы можно манипулировать с помощью указателей с адресами этих переменных — точно так же, как с помощью собственных имен этих переменных. В главе 6 уже рассказывалось о том, как использовать указатели для динамического управления памятью. Данный раздел посвящен концепциям, связанным с применением указателей для передачи параметров.

Указатели — мощный и гибкий инструмент программирования, но вместе с тем и опасный. Именно по этой причине в C++ использование указателей ограничивается. При определении указателя программист берет на себя некоторые обязательства: его указатель должен ссылаться на переменные типа `int`, `double`, `Account`, `Square` и пр., что не отличается от определения переменных других видов.

Во всех других отношениях указатели — это обычные переменные. Они имеют тип, могут инициализироваться, им присваиваются имена, к ним применяются операции. В конечном счете, переменные-указатели уничтожаются согласно правилам C++. При определении переменной указателя к ней добавляется звездочка (*). Тем самым помечается, что это именно указатель. При создании указателя он не содержит допустимого значения. Как и любую другую переменную, указатель нужно инициализировать или присваивать ему значение:

```
int v1, v2;                  // две целочисленные переменные;
                             // пока что содержат "мусор"
int *p1, *p2, *p3;           // указатели на целые; пока никуда не ссылаются
```

К указателям можно применять такие операции как присваивание, сравнение, разыменование. Разрешается присваивать им следующие значения:

- `NULL`
- Значение, содержащееся в другом указателе такого же типа (допускается его увеличение или уменьшение на целое число)
- Адрес переменной соответствующего типа (оператор получения адреса C++ — `&` — присоединяется слева к имени переменой, адрес которой присвоен указателю)

```
v1 = 123;  v2 = 456;     // переменным присваивается целое значение
p1 = &v1;                // указателю присваивается адрес переменной v1
p2 = p1;                 // указателю присваивается значение другого указателя
p3 = NULL;               // указатель на присвоенное значение NULL
```

Символьная константа `NULL` определена в заголовочном файле `stdlib.h` и во многих других библиотечных файлах, включая `iostream.h`. Это то же самое, что 0. Некоторые программисты предпочитают использовать `NULL`, поскольку тем самым явно указывается, что программа имеет дело с указателями. Другие программисты применяют 0, это тоже нормально. Обычно программистам с опытом работы на языке C больше нравится `NULL`, а программистам, привыкшим к C++,— значение 0.

И `p1`, и `p2` теперь указывают на переменную `v1`. Для сравнения указателей применяется тот же синтаксис, что и для сравнения любых других числовых значений.

```
if (p1 == p2) cout << "Одинаковый адрес, а не значения по адресу\n";
if (p3 == 0) cout << "Это нулевой указатель\n";
if (p1 != 0) cout << "Можно начинать работу\n";
```

Последняя операция, операция разыменования, обозначается как звездочка *. При применении к указателю она дает значение, на которое тот ссылается. Тип значения будет соответствовать типу, использованному при определении указателя.

Пусть p1 ссылается на целочисленную переменную v1, содержащую 123. Это целое. Та же звездочка использовалась при определении указателя: int *p1;, что не случайно. Здесь говорится не только, что p1 — указатель на целое, но и что *p1 — целое. Поэтому областью действия звездочки является только одно имя. При определении целочисленных переменных (или переменных любых других встроенных типов) имя типа охватывает любое число переменных. Например, выше переменные v1 и v2 определяются с помощью одного ключевого слова int. С указателями это не работает. Следующая строка определяет один указатель и две целочисленные переменные:

```
int* pt1, pt2, pt3;          // pt1 - указатель, pt2 и pt3 - целые
```

Вернемся к разыменованию. Разыменованный указатель — это всего лишь переменная, на которую он ссылается.

```
*p1 = 42;          // v1 теперь не 123, это 42
*p2 = 180;         // v1 теперь не 42, это 180
*p3 = 42;          // не разыменовывайте указателей NULL;
                   // это ведет к аварийному завершению программы.
```

В данном примере p1 ссылается на v1, следовательно, *p1 и v1 — во всех отношениях синонимы. До тех пор, пока указателю не будет присвоено что-то еще:

```
p1 = &v2;                    // p1 теперь ссылается на v2, а не на v1;
                             // а *p1 означает 456
if (*p1 == 456) *p1 = 42;    // v2 больше не 456, это 42
```

Если разыменованные указатели и переменные, на которые они ссылаются — синонимы, то зачем вообще использовать указатели в тех случаях, когда это не связано с динамическим распределением памяти? Дело вот в чем: мы будем применять параметры-указатели для изменения значений фактических аргументов в клиентской области. Если указатель-переменная передается функции, то функция может изменить значение, на которое он ссылается (фактический аргумент) с помощью разыменования.

Рассмотрим, например, следующую модификацию функции add(). Аналогично предыдущей версии, она вычисляет сумму двух аргументов. Вместо возврата результата она присваивает его разыменованному указателю, т. е. значению, на которое ссылается параметр-указатель:

```
void add (int x, int y, int *z)          // z - указатель на целое
{ *z = x + y; }              // указатель z теперь ссылается на другое место
```

Как такая функция add() должна вызываться в клиентской программе? Как получить присвоенное указателю значение? Это не может быть double, short или int. Оно должно быть равно NULL (что в данном случае бесполезно), быть равным другому указателю (возможно, в других примерах, но не здесь) или представлять адрес целого (т. е. то, что нужно). В качестве третьего аргумента передается адрес переменной, которая должна содержать сумму. Чтобы получить адрес, можно использовать операцию &. Вот как вызов с указателем должен выглядеть в клиентском коде:

```
int a = 2, b = 3, c;
add(a, b, &c)                // c не изменяется после вызова
```

Мы уже говорили, что вызов по значению — естественный режим передачи параметров в C++. Это относится и к случаю передачи по указателю. По значению передается указатель, а не значение в клиентской области. Как и в случае передачи по значению, в области действия функции-сервера создается локальная копия указателя, которая затем инициализируется и в конечном счете уничтожается.

Поскольку функции передается адрес фактического аргумента, можно разыменовать указатель в теле функции, когда она обращается к значению фактического аргумента. Новое значение, присвоенное разыменованной переменной в функции, сохраняется в области действия клиента.

Если это кажется несколько запутанным, не стоит беспокоиться. Вы освоитесь с данной концепцией. Запомните лишь, что при передаче по указателю нужно задать:

- Операцию адреса, примененную к фактическому аргументу в вызове
- Тип указателя для параметра в заголовке функции
- Операцию разыменования для параметров в теле функции

Следуйте данной логике, и все будет в порядке. Любое отклонение от этого перечня приведет к тому, что что-то пойдет не так. Не стоит тратить на это силы.

Рассмотрим еще один популярный пример: перестановку значений параметров. Если первый параметр больше второго, они меняются местами (переупорядочение в правильной последовательности — по возрастанию). Для перестановки значений параметров a1 и a2 значение первого сохраняется во временной переменной temp. Затем можно скопировать значение a2 в a1, а значение из временной переменной — в a2. То, что раньше было в a2, теперь окажется в a1. В листинге 7.1 показана реализация функции swap() и ее клиентской функции main(). В целях отладки сюда включены также операторы вывода значений на экран. Результаты выполнения представлены на рис.7.1.

```
Перед обменом значениями: a1=84 a2=42
После обмена значениями:  a1=42 a2=84
После вызова:             x=84 y=42
```

Рис. 7.1. *Вывод для программы из листинга 7.1*

Листинг 7.1. Передача параметров с побочными эффектами (плохая версия)

```
#include <iostream>
using namespace std;

void swap (int a1, int a2)          // неверный режим передачи параметров
{ int temp;
  if (a1 > a2)
  { cout << "Перед обменом значениями: a1=" << a1 << " a2=" << a2 << endl;
    temp = a1; a1 = a2; a2 = temp;
    cout << "После обмена значениями: a1=" << a1 << " a2=" << a2 << endl;}}

int main()
{
   int x = 84, y = 42;              // неупорядоченные значения
   swap(x,y);                       // неверный режим передачи параметров - не должно работать
   cout << "После вызова: x=" << x << y=" << y << endl;
   return 0;
   }
```

Как можно ожидать, обмен значениями параметров в самой функции происходит корректно, однако это не отражается в области действия клиента, где значения местами не меняются. Здесь должна помочь передача параметров по указателю. Вот как выглядит следующая версия функции swap():

```
void swap (int *a1, int *a2)          // корректный режим передачи параметров
{ int temp;
  if (a1 > a2)
  { cout << "Перед обменом значениями: *a1=" << a1 << " a2=" << a2 << endl;
    temp = a1; a1 = a2; a2 = temp;
    cout << "После обмена значениями: a1=" << a1 << " a2=" << a2 << endl; }}
```

Хорошо смотрится, но тоже не летает. Оператор temp = a1; некорректен. Переменная temp имеет тип int, а переменная a1 — нет. Это указатель на int. Одно другому присваивать нельзя. Но можно присвоить один указатель на целое другому указателю на целое. Не позволяйте своему опыту работы с числовыми типами ввести вас в заблуждение. Разные числовые типы совместимы. Их можно привести один к другому. Значения указателей — типы несовместимые, и такое преобразование не допускается. Их нельзя привести к типу, отличному от указателя.

Если вы получите такое сообщение, не отчаивайтесь, а просто подумайте, что должно быть в правой части присваивания. Переменная a1 — это не целое. Какая родственная переменная имеет целое значение? Посмотрите на список параметров. Где здесь целое? Да, int *a1, следовательно, *a1 и есть целое, а присваивание должно выглядеть так: temp = *a1. Фактически, это не очень трудно, однако внесение изменений в тело функции — задача утомительная и способствующая ошибкам. Нужно убедиться, что выполняется разыменование a1 и a2, но не используется разыменование temp:

```
void swap (int *a1, int *a2)          // корректный режим передачи параметров
{ int temp;
  if (a1 > a2) {
     cout << "Перед обменом: *a1=" << *a1 << " *a2=" << *a2 << endl;
     temp = *a1; *a1 = *a2; *a2 = temp;      // корректное разыменовывание
     cout << "После обмена: *a1=" << *a1 << " *a2=" << *a2 << endl; } }
```

В версии функции swap() в листинге 7.1 компилятор сообщает о вызове swap(x,y). Действительно, переменная x имеет тип int, т. е. это адрес целочисленной переменной. Есть ли здесь смысл? Корректная версия программы показана в листинге 7.2, а соответствующий вывод — на рис. 7.2.

```
Перед обменом значениями: *a1=82 *a2=42
После обмена значениями:  *a1=42 *a2=82
После вызова:             x=42 y=82
```

Рис. 7.2. *Вывод для программы из листинга 7.2*

Листинг 7.2. Передача параметров по указателю (режимы параметров корректны)

```
#include <iostream>
using namespace std;

void swap (int *a1, int *a2)                    // корректный режим передачи параметров
{ int temp;
  if (a1 > a2)
     { cout << "Перед обменом значениями: *a1=" << *a1 << " *a2=" << *a2 << endl;
     temp = *a1; *a1 = *a2; *a2 = temp;         // корректное разыменовывание
     cout << "После обмена значениями: *a1=" << *a1 << " *a2=" << *a2 << endl; }}

int main()
{
   int x = 82, y = 42                           // неупорядоченные значения
   swap(&x,&y);                                 // верный режим передачи параметров
   cout << "После вызова: x=" << x << "y=" y << endl;
   return 0;
   }
```

Сделано ли все необходимое для тестирования данной программы? Она содержит условный оператор, проверявшийся только один раз. Это простая маленькая программа, и стоит ли тратить время на проверку того, что и так ясно как божий день? Всего не проверишь. Тем не менее, когда дело касается передачи параметров, нельзя быть ни в чем уверенным (шаг влево, шаг вправо...). А потому изменим

функцию main() следующим образом, просто, чтобы убедиться, что когда аргументы следуют в верном порядке, функция не меняет местами их значение:

```
int main()
//{ int x = 82, y = 42                    // неупорядоченные значения
{ int x = 42, y = 84                      // неупорядоченные значения
  swap(&x,&y);                            // верный режим передачи параметров -
                                          // не должно работать
  cout << "После вызова: x=" << x << "y=" y << endl;
  return 0; }
```

```
Перед обменом значениями: *a1=42 *a2=84
После обмена значениями:  *a1=84 *a2=42
После вызова:             x=84 y=42
```

Рис. 7.3. *Вывод для программы из листинга 7.3*

Результат выполнения показан на рис. 7.3. Теперь придется задать два вопроса. Вопрос №1: видите ли вы проблему в результатах? Убедитесь, что это так. Слишком часто мы смотрим на результаты и не видим ошибку, так как заранее не записали ответ. Похоже, что программа переставляет аргументы безусловно, хотя в функции swap() имеется оператор if. Возможно, это означает, что этот оператор сравнивает не значения параметров, а что-то еще. Вопрос №2: видите ли вы проблему в исходном коде в листинге 7.2? Это уже труднее, поскольку нет методологии поиска данной ошибки.

Функция swap() содержит 10 операций *, но нужны еще две. При сравнении a1 и a2 компилятор не возражает, так как эти две переменные имеют один тип. Если требуется сравнить два адреса, на это у вас полное право, и если первый адрес больше второго, программа будет менять местами аргументы, независимо от их порядка. Компилятор не гадает за программиста, что именно нужно делать, и не выводит предупреждений. Корректная версия программы показана в листинге 7.3.

Листинг 7.3. Передача параметров по указателю (исправлено разыменование)

```
#include <iostream>
using namespace std;

void swap (int *a1, int *a2)                    // корректный режим передачи параметров
{ int temp;
  if (*a1 > *a2){                               // Ой-ой-ой
     cout << "Перед обменом значениями: *a1=" << *a1 << " *a2=" << *a2 << endl;
     temp = *a1; *a1 = *a2; *a2 = temp;         // корректное разыменовывание
     cout << "После обмена значениями: *a1=" << *a1 << " *a2=" << *a2 << endl;}}

int main()
// { int x = 82, y = 42                         // неупорядоченные значения
{ int x = 42, y = 84;                           // упорядоченные значения
  swap(&x,&y);                                  // верный режим передачи параметров
  cout << "После вызова: x=" << x << "y=" << y << endl;
  return 0;
  }
```

Кстати, это не абстрактный пример, а уменьшенная версия вполне реальной задачи. В ней для простоты намеренно убраны лишние детали.

Мнемоническое правило при передаче параметров состоит в следующем: при выборе имени параметра нужно начать с этого имени со звездочкой и не забывать о звездочке при использовании имени во всех других случаях. Проблема в первой версии swap(), да и во второй тоже, состояла в том, что в качестве имен параметров использовались a1 и a2. Если бы с самого начала были выбраны и везде использовались *a1 и *a2, то написание такой функции не представляло бы труда, особенно, если применить в операторе условия выражение *a1 > *a2.

Но это сложнее, чем передача параметров по значению. Программисту нужно:

- Использовать обозначение указателя в заголовке функции (и прототипе)
- Разыменовывать указатели в теле функции
- Применять операцию получения адреса вне функции (в коде клиента). В награду он получает "побочный эффект" — изменения в параметрах отражаются в фактических аргументах в клиентском коде

Каждому случается делать ошибки при передаче параметров по указателю. Разница между опытными и неопытными программистами в том, что опытные делают эти ошибки реже и исправляют быстрее. Но не стоит увлекаться самокритикой. В конце концов, не вы изобрели эти правила.

Некоторые программисты пытаются упростить передачу параметров по указателю, исключив использование операции получения адреса при вызове функции. Вместо этого они применяют указатели, которым присвоена ссылка на фактические аргументы. Например, вызов функции swap() можно записать следующим образом:

```
int main()
{ int x = 82, y = 42;                  // неупорядоченные значения
  int *p1 = &x, *p2 = &y;              // использование указателей для ссылки
                                       // на значения
  swap(p1,p2);                         // нет операции получения адреса
  cout << "После вызова: x=" << x << " y=" << y << endl;
  return 0; }
```

Это работает. Значения аргументов, на которые ссылаются p1 и p2, будут корректно меняться местами. Однако приходится вводить в программу дополнительные элементы, что увеличивает вероятность ошибок и усложняет понимание программы. Не факт, что данный метод проще, чем применение операции получения адреса непосредственно в вызове функции, но он вполне работоспособен.

Поскольку при вызове по указателю значения фактических аргументов меняются, фактическими аргументами могут быть только l-значения, которыми можно манипулировать через адреса. Применение r-значений — выражений, литеральных значений или констант — не допускается. Например, такой вызов функции swap() некорректен:

```
swap(&5, &(x+y));              // не годится: для r-значений
                               // нет операции получения адреса
```

Еще один вопрос, связанный с передачей параметров по указателю,— преобразование типов. Оно не допускается в любом случае. То, о чем говорилось выше, относится к преобразованию типов значений, а не указателей. Рассмотрим следующий пример и попытаемся использовать функцию swap() для упорядочения значений типа double:

```
int main()
{ double x = 82, y = 42;               // значения double не упорядочены
  swap(&x,&y);                         // нет преобразования из double* в int*
  cout << "После вызова: x=" << x << " y=" << y << endl;
  return 0; }
```

Можно попытаться "уговорить" компилятор принять эти аргументы, применив в вызове функции явное приведение типа к int*:

```
swap((int*)&x, (int*)&y);              // будет менять местами значения int,
                                       // а не double
```

Такой вызов будет компилироваться, но здесь сравниваются и меняются местами (если это происходит) только части значений double, имеющие размер целого. Компилятор проглотил это, так как программист сказал ему: "Я знаю, что делаю". Однако то, что он делает, некорректно. Убедитесь, что при компиляции программы воспринимаемое компилятором действительно имеет смысл.

Поскольку в C++ явным образом поставлена цель поддерживать унаследованный код на языке C, передача параметров по указателю является вполне законным методом C++. В C++ добавлен еще один режим передачи параметров — передача по ссылке, в которой устранены некоторые недостатки передачи по указателю. Мы попытаемся применять передачу по указателю по возможности реже. К сожалению, просто забыть о ней нельзя. Кроме унаследованных программ на языке C существуют библиотечные функции C++, параметры которым передаются по указателю. Управление динамическим распределением памяти также требует работы с указателями. Так что сложность операций с указателями не должна пугать вас.

Передача параметров в C++ по ссылке

Кроме указателей C++ предусматривает ссылочные типы, недоступные в языке C. Как и переменная-указатель, ссылка содержит адрес другой переменной. Подобно указателям, разрешается определять ссылки любого встроенного или созданного программистом типа. Как и указатели, переменные-ссылки — это обычные переменные, которые можно определять, распределять для них память, инициализировать и уничтожать.

В отличие от указателя, переменная ссылочного типа может ссылаться только на один адрес памяти, причем того же типа, что и сама ссылка. Переменную-ссылку нельзя переназначить на другой адрес памяти, поэтому, в отличие от указателей, переменные-ссылки должны инициализироваться в определении. Если не сделать этого, то переменная ссылка вообще не будет никуда ссылаться и станет бесполезной. Чтобы указать, что переменная является ссылкой, а не указателем, слева от ее имени помещается амперсанд (&), а не звездочка, как в указателе. После инициализации ссылки к переменной, на которую эта ссылка указывает, не нужно применять никакие операции. Такое изменение в обозначении весьма разумно. Это одна из причин введения ссылок в C++.

```
int v1=123,  v2=456;          // целочисленные переменные, не обязательная
                              // инициализация
int *p1=&v1, *p2=&v2;         // указатели на int, не обязательная инициализация
int &r1=v1, &r2=v2;           // ссылки: всегда инициализируются, нет операций
```

Для указателей *p1 и v1 — синонимы. Нужна операция разыменования *. Для ссылок синонимами являются r1 и v1, и никаких операций не требуется. В C++ для ссылок нет операции разыменования, и это еще одна причина введения ссылок.

```
if (p1 != p2) cout << "Разные адреса\n";          // конечно, &v1 != &v2
if (*p1 != *p2) cout << "Разные значения\n";      // конечно, 123 != 456
if (r1 != r2) cout << "Разные значения\n";        // конечно, 123 != 456
```

При работе с указателями нет разницы, как получать значение — с помощью операции разыменования (например, *p1) или имени переменной (такого, как v1). При использовании ссылок все равно, как вы ссылаетесь на значение: по имени ссылки без какой-либо операции (например, r1) или по имени переменной (v1). Это синонимы:

```
*p1 = 42;          // v1 (и r1) содержат теперь не 123, а 42
r1 = 180;          // v1 (и *p1) содержат теперь не 42, а 180
v1 = 42;           // r1 (и *p1) содержат снова 42
```

Возможно, это кажется несколько запутанным: указатели "разыменовываются", а не "разуказываются". С другой стороны, нельзя разыменовывать ссылки. Все придумано вовсе не для того, чтобы вас запутать. Просто так исторически сложилось. Еще до появления ANSI C указатели часто называли ссылками, поскольку они ссылаются на переменные, а передачу по указателю называли передачей по ссылке. Вместо термина "разуказание" (depointing) всегда использовался термин "разыменование" (dereferencing). Такие названия сохранились и доныне.

Данная терминология пережила стандартизацию ANSI. При разработке C++ потребовались новые термины для таких понятий, как ссылки, доступ, указание, обозначение и пр. В процессе отбора был выбран термин "ссылка" (reference), так что теперь мы разыменовываем указатели, а не ссылки.

В отличие от указателя после инициализации ссылка не может изменяться для указания на другую переменную (адрес). Никаких способов для этого не предусматривается — адрес и ссылка неразделимы "до самой смерти", т. е. до завершения области действия. Присваивание ссылке приводит к изменениям самих данных, а не их адреса, т. к. ссылка не предполагает псевдонимов для переменной.

```
p1 = &v1;              // p1 теперь ссылается не на v1, а на v2
p1 = p2;               // еще один способ - тот же результат
r1 = v2;               // r1 все равно указывает на v1, где теперь содержится 456
r1 = r2;               // еще один способ переместить данные из v2 в v1
if (r1==v2) r1 = 42;    // сравнение дает true, и v1 теперь содержит 42
```

Это все, что нужно знать о ссылках, их терминологии и обозначениях, применяемых для передачи параметров. Вот как выглядит функция add() при передаче третьего параметра по ссылке, а не по указателю:

```
void add(int x, int y, int &z)      // z - ссылка на целое
{ z = x + y; }                      // изменяются данные, на которые ссылается z
```

Здесь переменная z — ссылка на целое. Когда вызывается функция, для данного параметра выделяется память, и он инициализируется адресом фактического аргумента. (Чуть позднее мы увидим, как это делается.) Присваивание z модифицирует данные, на которые указывает ссылка, т. е. фактический аргумент. Видно, что тело функции выглядит в точности так же, как если бы параметр z передавался по значению. Применять разыменование нет необходимости. В заголовке функции & указывает на передачу по ссылке. В вызове функции нужно инициализировать ссылку адресом, на который она ссылается. Как это сделать? Согласно синтаксису инициализации переменной, можно использовать имя переменной без операций. Следовательно, вызов функции в клиентском коде должен выглядеть так:

```
int a = 2, b = 3, c;
add(a, b, c);                   // c не изменяется после вызова
```

При передаче параметров по ссылке нужно задавать:

- Имена аргументов без операции получения адреса в вызове
- Тип ссылки для параметров в заголовке функции (и прототипе)
- Имена параметров без разыменования в теле функции

Как видно, вызов по ссылке очень похож на вызов по значению. Параметры в теле функции не разыменовываются, а в вызове функции в клиентском коде операции получения адреса не требуется. Тем не менее благодаря операции ссылки в заголовке функции-сервера, побочные эффекты на аргументы имеют место.

По существу эта конструкция напоминает язык Паскаль, где ключевое слово var играет роль операции & в C++. Она указывает, что формальные параметры изменяются в теле функции, и эти изменения отражаются в значениях фактических

```
Перед обменом значениями: a1=82 a2=42
После обмена значениями:  a1=42 a2=82
После вызова:             x=42 y=82
```

Рис. 7.4. *Вывод для программы из листинга 7.4*

аргументов в программе-клиенте. Возможно, ключевое слово проще и легче запоминается, чем операция.

Давайте теперь реализуем функцию перестановки значений аргументов с помощью передачи по ссылке. Исходный код показан в листинге 7.4. Изменения достаточно просты. Исходный код также стал намного понятнее, чем при применении указателей. Уменьшилась вероятность ошибок. Результаты представлены на рис. 7.4.

Листинг 7.4. Передача параметров по ссылке (надежный метод)

```
#include <iostream>
using namespace std;

void swap (int &a1, int &a2)                    // корректный режим передачи параметров
{ int temp;
  if (a1 > a2) {                                // разыменования не требуется
  cout << "Перед обменом значениями: a1=" << a1 << " a2=" << a2 << endl;
  temp = a1; a1 = a2; a2 = temp;
  cout << "После обмена значениями: a1=" << a1 << " a2=" << a2 << endl;}}

int main()
{ int x = 82, y = 42                            // неупорядоченные значения
// { int x = 42, y = 84;                        // упорядоченные значения
  swap(x,y);                                    // верный режим передачи параметров
  cout << "После вызова: x=" << x << " y=" << y << endl;
  return 0;
  }
```

В таблице 7.1 отображены правила передачи параметров. Здесь термин var (с операциями, там, где это применимо) обозначает имя переменной, используемой в качестве аргумента в вызове функции, параметра в заголовке функции (и прототипе) и в теле функции.

Таблица показывает простейший случай передачи значения, когда к аргументу или к параметру в теле функции не применяются операции. Передача по указателю более сложна, так как в трех местах нужны операции: в аргументе, в параметре и в теле функции. Передача по ссылке аналогична передаче по значению. Единственная разница — операция ссылки в заголовке функции.

При передаче по ссылке нет свойственной передаче по указателю сложности, но поддерживается изменение аргументов в области действия клиента. Если параметры в теле функции не изменяются, их следует передавать по значению. Тогда всегда будет понятно, что разработчик хотел сохранить значения фактических аргументов.

Таблица 7.1

Передача параметров в случае простой переменной

Элемент исходного кода	По значению	По указателю	По ссылке
Вызов функции	var	&var	var
Заголовок функции	var	*var	&var
Тело функции	var	*var	var

Когда параметры передаются по ссылке и модифицируются внутри функции, действуют те же ограничения, что и при передаче по указателю, в частности,

l-значения можно использовать только как фактические аргументы, они должны быть одного типа, поскольку C++ не поддерживает неявного преобразования ссылок на разные типы. Явное преобразование возможно, но бесполезно, так как переменная-ссылка позволяет обращаться лишь к значению того типа, которое было задано в определении. Применение выражений, литералов и констант не допускается.



Советуем Когда функция не изменяет значений своих параметров встроенных типов C++, передавайте параметры по значению. Если в функции нужно изменять значения параметров встроенных типов, передавайте их по ссылке. Избегайте передачи по указателю.

Структуры

Структуры (и объекты классов) можно передавать по значению, по указателю или по ссылке. Если переменная-структура используется функцией "на входе" и не изменяется в ее теле, ее можно передавать по значению. Если структура является выходным значением функции (т. е. служит для передачи значений функции-клиенту), и в теле функции модифицируются ее поля, то следует передавать ее по указателю или по ссылке. В противном случае эти изменения не будут действовать в пространстве клиента.

Правила, сформулированные в предыдущем разделе для отдельной переменной, применимы и к структуре. Дополнительные правила, действующие при использовании структур в качестве аргументов, относятся к доступу к компонентам структуры в теле функции.

Для простоты в качестве примера рассмотрим упрощенный тип Account:

```
struct Account {
  long num;                    // для простоты только два поля
  double bal; } ;
```

Функция доступа к этой структуре printAccounts() имеет два параметра-переменных типа Account и выводит два значения: номер счета и остаток на счете. Эти объекты Account являются для printAccounts() исходными переменными. Их значения должны устанавливаться в клиентском коде перед вызовом, так как функции printAccounts() для правильной работы нужны уже установленные значения переменных типа Account:

```
void printAccounts(Account a1, Account a2)            // код сервера
{ cout  << "Номер первого счета: No. " << a1.num
        << "Остаток: " << a1.bal << endl;
  cout  << "Номер второго счета: No. " << a2.num
        << "Остаток: " << a2.bal << endl; }
```

В клиентском коде создаются объекты Account, инициализируются их поля и вызывается серверная функция printAccounts() для вывода состояния счетов:

```
Account x, y;                                          // код клиента
x.num = 800123456; x.bal = 1200;
y.num = 800123123; y.bal = 1500;
printAccounts(x,y);
```

Поскольку мы сейчас обсуждаем передачу параметров, не будем вдаваться в такие вопросы, как целесообразность применения функции для доступа к такой тривиальной структуре, или возможность прямого доступа клиента к полям структуры. Эти простые примеры наглядно иллюстрируют коммуникации между функциями.

Здесь действуют базовые правила передачи параметров, т. е. программисту нужно координировать код в трех разных местах: в вызове функции, в заголовке функции и в ее теле. Согласно правилам передачи параметров по значению, при вызове функции, в ее заголовке и теле используется имя переменной без операций. Когда в теле функции нужно обращаться к полям структуры, применяется та же операция-точка (селектор), что и в клиентском коде. Это простейший режим передачи параметров. (Сравните обозначение `a1.bal` в функции `printAccounts()` и `x.bal` в клиенте.)

Теперь рассмотрим другую функцию доступа — `swapAccounts()`, которая сравнивает номера счетов в своих параметрах и меняет параметры местами, если номера не упорядочены. Поскольку значения фактических аргументов в этом случае должны изменяться, передача параметров по значению уже не подходит. Эта функция передает свои параметры по указателю:

```
void swapAccounts(Account *a1, Account *a2)     // передача по указателю
{ Account temp;
  if (a1->num > a2->num)                        // операция
  { temp = *a1, *a1 = *a2; *a2 = temp; } }
```

Когда клиентский код вызывает эту функцию, он передает ей адреса фактических аргументов.

```
swapAccounts(&x,&y);
```

Как можно видеть, основные правила передачи по указателю применимы и здесь. В вызове функции клиент использует операцию получения адреса &, а в сервере в заголовке функции указывается звездочка *. В теле функции-сервера применяется операция разыменования *. Когда код сервера должен обращаться к полям структуры, вместо операции-точки используется двухсимвольный селектор-стрелка.

Это общее правило не ограничивается передачей параметров. Операция точки позволяет выбрать поле, когда левый операнд задает имя переменной-структуры. Операция-стрелка применяется в том случае, когда ее левым операндом является указатель на структурную переменную. Не путайте эти две операции. Для программистов различие часто очевидно. Когда используется указатель, не важно, на что он ссылается — на именованную переменную в стеке или на неименованную переменную в динамически распределяемой области. Для указателя нужна операция-стрелка. Если одна операция применяется в контексте, где требуется другая операция, генерируется сообщение об ошибке (иногда достаточно непонятное).

Некоторые программисты пытаются использовать соглашения по именам, позволяющие им визуально видеть, что переменная является указателем, а не значением. Они начинают имена указателей с `p` или `ptr`. Вот как будет выглядеть функция `swapAccounts()`, если соблюдать такое соглашение:

```
void swapAccounts(Account * ptrA1, Account *ptrA2)
                                                  // передача по указателю
{ Account temp;
  if (ptrA1->num > ptrA2->num)                    // операция
  { temp = * ptrA1, * ptrA1 = * ptrA2; * ptrA2 = temp; } }
```

На мой взгляд, предыдущая версия лучше, так как имена параметров (типа `Account`) вида `*a1` и `*a2` привычнее (для меня они начинаются со звездочки), а компонент `ptr` в имени малопривлекателен. Тем не менее подобная практика общепринята, и можно следовать ей при работе с указателями.

Для некоторых программистов необходимость выбора того или иного селектора — дополнительная проблема. Операцию точки можно использовать с указателем, если сначала нужно разыменовать его. Вот пример функции `swapAccount()`, где применяется этот метод:

```
void swapAccounts (Account *a1, Account *a2)          // передача по указателю
{ Account temp;
  if ((*a1).num > (*a2).num)                          // нет селектора-стрелки
  { temp = *a1; *a1 = *a2; *a2 = temp; } }
```

Круглые скобки здесь важны, так как операция-селектор имеет более высокий приоритет, чем разыменование. Выражение без скобок, например, `*a1.num` будет понято компилятором не как `(*a1).num`, а как `*(a1.bal)`, что не имеет смысла. Во-первых, выражение `*a1.bal` незаконно, так как указатель `a1` не может работать с селектором-точкой. Во-вторых, даже если бы выражение `a1.bal` было допустимым, поле `bal` имеет тип `double`, а разыменовать значение `double`, понятно, нельзя — это должен быть указатель.

Применение разыменования и селектора-точки — метод вполне законный, но большинство программистов чувствуют себя вполне комфортно, переключаясь с одной операции-селектора на другую, и не нуждаются в единообразии. Если все время использовать операцию-стрелку, могут подумать, что вы знаете C++ вовсе не так хорошо, как утверждаете.

В функции `swapAccounts()` структуры передаются по указателю, а не по значению, поскольку фактические аргументы должны изменяться в результате перестановки. Некоторые программисты, особенно, имеющие солидный опыт работы на языке C, не любят передавать структуры по значению и передают их по указателю, даже когда исходные переменные в процессе выполнения функции не изменяются. Вот как могла бы выглядеть функция `printAccounts()`, если бы ее написал такой программист. Параметры здесь передаются по указателю, а не по значению, а для обращения к полям структуры используется селектор-стрелка:

```
void printAccounts(Account *a1, Account *a2)          // вводит в заблуждение
{ cout << "Номер первого счета: No. " << a1->num
       << "Остаток: " << a1->bal << endl;
  cout << "Номер второго счета: No. " << a2->num
       << "Остаток: " << a2->bal << endl; }
```

Передавать параметры-структуры по указателю бывает желательно из соображений повышения производительности программы. Конечно, в этих примерах структура `Account` мала, но нередко приходится иметь дело со структурными объектами размером в сотни и тысячи байтов. Копирование таких структур при передаче по значению требует времени и памяти, что бывает существенно для быстродействия программы. Недостатком передачи по указателю является усложнение программы. Кроме того, некоторые программисты видят опасность несанкционированного изменения фактических аргументов или случайной порчи их данных. Когда параметры передаются по значению, даже при попытке модифицировать их в функции-сервере, изменения не подействуют на фактические аргументы, а при передаче параметров по указателю изменения, вносимые функцией-сервером, распространяются и на область действия клиента.

Вряд ли целостность данных можно считать здесь серьезным вопросом. Ведь если функция-сервер попытается некорректно изменить параметры, это будет обнаружено и исправлено, и не стоит полагаться на передачу параметров по значению.

Более существенный вопрос — понятность намерений разработчика. Разработчик не хочет изменять аргументы функции `printAccounts()`. Между тем, он скрывает это знание от программиста, сопровождающего приложение, так как в заголовке функции ясно говорится, что возможны изменения, ведь параметры передаются по указателю. То же самое получается, когда в вызове функции аргументы передаются по указателю, поскольку здесь используется операция получения адреса:

```
printAccounts(&x,&y);              // ясно, аргументы можно изменять!
```

В данном случае легко проверить четыре строки исходного кода и понять, что именно имеет место в действительности, но когда требуется просматривать много строк, все значительно осложняется, тем более, что эти строки могут содержать нечто весьма туманное. В случае же передачи параметров по значению нет никакой необходимости анализировать тело функции. Это очень важный вопрос.

Но производительность также важна. Передача параметров по ссылке дает возможность выбрать компромиссный вариант: и овцы целы, и волки сыты. Она позволяет преодолеть сложности передачи по указателю и потери производительности при передаче по значению. При этом четко видно намерение разработчика.

Как же быть? В листинге 7.5 приведен пример программы, реализующей обе серверные функции — printAccounts() и swapAccounts() — путем передачи параметров по ссылке. Результаты программы представлены на рис. 7.5.

```
Перед перестановкой
Номер первого счета: 800123456  Остаток: 1200
Номер второго счета: 800123123  Остаток: 1500
После перестановки:
Номер первого счета: 800123123  Остаток: 1500
Номер второго счета: 800123456  Остаток: 1200
```

Рис. 7.5. *Вывод программы из листинга 7.5*

Листинг 7.5. Пример передачи параметров по ссылке

```
#include <iostream>
using namespace std;

struct Account {
   long num;
   double bal; } ;

void printAccounts(cons Accounts &a1, const Account &a2)          // заголовок
{ cout << "Номер первого счета: No. " << a1.num                  // тело
       << "Остаток: " << a1.bal << endl;
  cout << "Номер второго счета: No. " << a2.num
       << "Остаток: " << a2.bal << endl; }

void swapAccounts(Account &a1, Account &a2)                       // заголовок
{ Account temp;                                                   // тело
  if (a1.num > a2.num)
  { temp = a1; a2 = a2; a2 = temp; } }

int main()
{
   Account x, y;
   x.num = 800123456; x.bal = 1200;
   y.num = 800123123; y.bal = 1500;
   cout << "Перед перестановкой\n";
   printAccounts(x,y);                                            // вызов
   swapAccounts(x,y);                                             // вызов
   cout << "После перестановки\n";
   printAccounts(x,y);
   return 0;
   }
```

Функция swapAccounts() достаточно проста. В вызове функции используется имя структуры, обозначение ссылки применяется в заголовке функции, и имя структуры — в ее теле. Для доступа к полям в теле функции служит операция-точка, что опять поднимает вопрос выбора корректного селектора. Как видно, при передаче параметров-структур по ссылке обозначения проще, чем при передаче по указателю.

Функция printAccounts() также проста. В вызове функции указывается имя структурной переменной (как при передаче по значению), а в теле функции используются имена параметров. Доступ к полям структуры осуществляется точно так же, как при передаче по значению. Все различия — в заголовке функции: здесь с именами параметров используется обозначение ссылки и ключевое слово const перед типами параметров. Первое исключает копирование фактических аргументов (функции передаются их адреса, а не копии полей), а второе предотвращает изменение значений параметров в функции и ясно говорит о том, что параметры не изменяются. Нет необходимости проверять тело функции, чтобы убедиться в этом.

Применение ключевого слова (модификатора) const аналогично его использованию в определении переменных. Данный модификатор можно применять со значениями, указателями и ссылками. Когда он указывается со значениями, то говорит о том, что значение не может изменяться ни путем прямого присваивания, ни через указатель или ссылку.

```
cosnt int val = 10; // инициализация обязательна
val = 20;           // синтаксическая ошибка: присваивание не допускается
int *p = &val;      // не допускается для запрета косвенного изменения *p = 20;
int &r = val;       // не допускается для запрета косвенного изменения r = 20;
```

Когда модификатор const используется с указателями и ссылками, у него в зависимости от позиции может быть два смысла. Если он указывается перед именем типа, это значит, что значение не может модифицироваться путем разыменования указателя или через ссылку:

```
const int val = 10;
const int *constp = &val;      // OK, но *constp=20 даст синтаксическую ошибку
const int &constp = val;       // OK, но constr=20 даст синтаксическую ошибку
```

Обратите внимание, что непосредственное изменение переменной val, например val = 20, не допускается, поскольку данная переменная определена как константа. Косвенная модификация также недопустима, но не потому, что "переменная" val — константа, а потому, что переменная-указатель (и ссылка) определена как указатель на const. Для указателя (или ссылки) на константу косвенная модификация не допускается, даже при ссылке на "не константу". Почувствуйте разницу:

```
int value = 10;                    // эта переменная может изменяться
const int *constp = &value;        // *constp=20 все равно даст
                                   // синтаксическую ошибку
const int &constr = value;         // *constr=20 все равно даст
                                   // синтаксическую ошибку
```

Когда модификатор const находится между именем типа и именем указателя, это означает, что указатель является константой. К нему можно применить разыменование, а значение, на которое указатель ссылается, можно изменить, однако не допускается присваивание указателю другого адреса. Хотя инициализация не обязательна, она необходима. Если указатель-константу не инициализировать при определении, позднее ему уже ничего нельзя будет присвоить:

```
int value = 10;                    // в данном примере это не константа
int* const pconst = &value;        // теперь навеки вместе
*pconst = 20;                      // OK, значение - не константа
pconst = NULL;                     // синтаксическая ошибка:
                                   // указатель - константа
```

Ссылочную константу объявить нельзя, так как ссылки в C++ являются константами по умолчанию. Они инициализируются в определении и не могут указывать на другой адрес. Используемое в передаче параметров обозначение говорит

о том, что ссылку не только нельзя переустановить на другой адрес, но и этот адрес не может изменять своего значения. Данная ссылка инициализируется значением фактического аргумента во время вызова функции.

Что происходит, если разработчик устает анализировать все эти тонкости и передает структуру по ссылке без модификатора const? Ничего особенного. Вот упрощенная функция printAccounts():

```
void printAccounts(Account &a1, Account &a2)          // можно изменять?
{ cout << "Номер первого счета: " << a1->num
       << "Остаток: " << a1->bal << endl;
  cout << "Номер второго счета: " << a2->num
       << "Остаток: " << a2->bal << endl; }
```

Корректная ли эта функция? Да. К тому же, обещая не изменять свои параметры, она их не изменяет. Тем не менее это несомненный вклад в кризис ПО. Разработчик не сообщил, что у него не было намерений модифицировать параметры в теле функции.

Некоторые программисты считают, что применение модификатора const в теле функции полезно, так как защищает аргументы от несанкционированного изменения в функции. Но дело не в том, что при вызове функции случаются ошибки, а в том, что при анализе исходного кода функции желательно знать, что она делает со своими параметрами, и модификатор const четко говорит, что параметры используются для ввода. Отсутствие же модификатора const может свидетельствовать о том, что функция изменяет параметр, а не о том, что программист устал учитывать все эти тонкости.

Возьмите за правило: если параметр-структура изменяется в функции, передавайте его по ссылке и не применяйте ключевое слово const. Если же параметр-структура не изменяется, передавайте его по ссылке без const.

Аналогично, отсутствие модификатора const должно ясно говорить сопровождающему приложение программисту, что параметры меняются в теле функции, а не о том, что разработчик рассеян и забывчив. Звучит довольно витиевато, но в C++ нет другого способа различать входные и выходные параметры.



Советуем Чтобы избежать отрицательного влияния на производительность, не передавайте структуры по значению, всегда передавайте их по ссылке. Когда параметр не модифицируется функцией, укажите это с помощью ключевого слова const. Когда параметр изменяется функцией, обозначьте это тем, что const не используется.

Передача структуры по ссылке имеет преимущество при сравнении с передачей по значению и по указателю. Она позволяет получить высокую скорость (без копирования данных) и добиться простоты (без операции адреса в вызове и без разыменования в теле функции). При корректном использовании это дает разработчику возможность сообщить о своих намерениях. Передача по ссылке очень популярна в C++. Используйте ее корректно.

Массивы

Подобно передаче по указателю, массивы всегда передаются в специальном режиме массивов. Хотя их обозначение аналогично передаче по указателю, это не одно и то же. Если функция-сервер изменяет компоненты параметра-массива, изменения будут видимы в аргументе-массиве в области действия клиента.

Это единственный режим передачи массивов как параметров, доступный в C++. Его нужно использовать и для входных, и для выходных параметров. Как и в других случаях передачи параметров, необходимо координировать вызов функции, ее заголовок и тело функции.

Приведем пример функции, копирующей содержимое своего второго аргумента-массива в первый аргумент-массив. Так как функции не известен размер массивов, число копируемых компонентов задается третьим параметром.

```
void Copy(double dest[], double src[], int size)
  { for (int i=0; i < size; i++)                    // классический цикл
    dest[i] = src[i]; }
```

Конечно, вызывающая сторона должна обеспечить достаточно компонентов для данной операции. C++ не дает программисту защиты от порчи содержимого памяти. Вот пример клиентского кода:

```
double x[100], y[100]; int n=0;
do {
  cout << "Введите данные (0 для завершения): ";
  cin >> y[n++];                    // заполнение массива y[], присваивание n
    ... } while (true);
Copy(x,y,n);                        // копирование n компонентов y[] в x[]
```

В данном примере показано, что при передаче массивов задается:

- Имя массива без квадратных скобок в вызове функции
- Пустые квадратные скобки после имени массива в заголовке функции
- Компоненты массива (или имя массива без квадратных скобок) в теле функции

В отличие от передачи параметров по указателю в вызове функции в клиентском коде нет операции получения адреса, а в заголовке функции не используются обозначения указателей. Программисты, предпочитающие подчеркивать сходство между массивами и указателями, могли бы записать эту функцию так:

```
void Copy(double *dest, double *src, int size)
  { for (int i=0 i < size; i++)             // классический цикл для массива
    dest+i = src+i; }
```

Можно также разыменовывать адреса компонентов массива, что подобно (но не идентично) передаче параметров по указателю.

```
void Copy(double *dest, double *src, int size)
  { for (int i=0 i < size; i++)             // классический цикл для массива
    *(dest+i) = *(src+i); }                 // или *dest++ = *src++;
```

Такое обозначение ближе к передаче по указателю. Обозначение указателя используется в заголовке функции. Тем не менее в теле функции разыменовывается не параметр, а сумма параметра и индекса. Программисты, любящие подчеркивать подобие массивов и указателей, могут использовать в вызове функции адрес первого компонента массива:

```
double x[100], y[100]; int n;     // заполнение массива y[], присваивание n
Copy(&x[0],&y[0],n);              // копирование n компонентов массива y[] в x[]
```

Тем не менее это адрес первого компонента массива, а не адрес фактического аргумента.

Какой бы синтаксис ни использовался, ни вызов, ни заголовок функции не могут указать на роль параметров (входные они или выходные). В данном примере массив scr[] является входным параметром (исходный массив). Значения его компонентов используются в теле функции для выполнения операций, а не являются результатом вызова. Массив dest[] — выходной параметр (результирующий массив). Какие бы значения ни содержали его компоненты перед вызовом,

в результате вызова они будут изменены. Обозначение обоих массивов во всех трех местах (вызове функции, в ее заголовке и в теле) одинаково. Это неправильно. Когда разработчик не планирует изменять массив в функции, лучше использовать модификатор const, чтобы сообщить об этих намерениях:

```
void Copy(double dest[], const double src[], int size)
  { for (int i=0; i < size; i++)
    dest[i] = src[i]; }          // src[i] = dest[i]; синтаксическая ошибка
```

Как и при передаче параметров-структур, очень важно использовать при передаче параметров-массивов, не изменяемых в теле функции, модификатор const, и делать это постоянно. Тем самым предотвращается изменение в функции исходных переменных (ввод) и, что еще важнее, ясно видны намерения разработчика.

Если параметр помечен как const, вместо него может подставляться фактический аргумент, помеченный или не помеченный как const. Отличный от const аргумент может изменяться, но функция не изменяет его, и это нормально. Но передавать переменную const как фактический аргумент, подставляя его в параметре, который не имеет модификатора const, нехорошо:

```
const double c[] = { 1.1, 1.2, 1.3, 1.3 } ;
Copy(x,c,4); Copy(y,a,4);              // OK, c[] и src[] - массивы const
Copy(c,x,4);      // синтаксическая ошибка: c[] - массив const, а dest[] - нет
```

Иногда использование модификаторов const затрудняет жизнь разработчику. Не сдавайтесь. Применяйте их. Как уже говорилось, современные языки усложняют написание исходного кода, но за счет этого упрощают его чтение. Нужно лишь убедиться, что действие и заявленное намерение совпадают и что параметры передаются фунциям-серверам согласованно.

Вот пример такой сложности. Некогда я написал эту простую функцию, вычисляющую сумму заданного числа компонентов массива:

```
double sum (double a[], int n)
{ double total = 0.0;                  // инициализировать подсчет
  for (int i = 0; i < n; i++)          // еще один классический цикл
    total += a[i];                     // накопление суммы
  return total; }
```

Позднее мне понадобилось подсчитать среднее по действительным элементам массива. Для этого нужно вычислить сумму и разделить ее на число элементов. Нетрудно было бы сделать все сначала, но, поскольку функция sum() уже есть, я решил ее использовать:

```
double avg (const double a[], int n)
  { return sum(a,n)/n); }                  // синтаксическая ошибка
```

Это пример передачи параметра через другую серверную функцию sum(). Предыдущие примеры показывают использование элементов параметра-массива и индекса в квадратных скобках. Данный пример демонстрирует, как использовать имя массива в теле функции без квадратных скобок.

Основная идея примера в том, что компилятору такой подход не нравится. Вот логика компилятора: массив a[] объявлен как const в заголовке функции avg(), следовательно, он не будет модифицироваться внутри avg(). Однако в теле avg() массив a[] передается функции sum() как аргумент, эта функция не принимает на себя никаких обязательств и вполне может изменить массив, нарушая то, что прописано в функции agv().

Компилятор не проверяет, изменяет ли функция sum() массив на самом деле. Но лучше перестраховаться, поэтому компилятор пометит функцию sum() как синтаксическую ошибку.

По идее, лучше бы компилятор проверял, что функция sum() делает со своим параметром. Однако не забывайте, что эта функция может находиться в другом файле и компилятор видит только ее прототип. Даже если она находится в том же файле, что и функция avg(), компилятор не анализирует поток значений в программе.

Чтобы исправить ситуацию, я определил все параметры как const. Намерения программы должны отражаться в интерфейсе функции-сервера:

```
double sum (const double a[], int n)                    // массив на входе
  { double total = 0.0;
    for (int i=0; i<n; i++)
       total += a[i];
    return total; }
```

В прототипе функции с параметрами-массивами соблюдаются правила, действующие для других прототипов функций. Заголовок функции завершается точкой с запятой:

```
double sum (const double a[], int n);                   // прототип функции
```

Имена параметров в прототипе не обязательны. Как это работает для массивов? Выглядит все забавно, но компилятор понимает такую запись. Пусть и вас она не смущает:

```
double sum (const double[], int);              // имена параметров не обязательны
```

Если для массивов применить указатели, то прототип будет выглядеть так:

```
double sum (const double*, int);               // указываемое значение не изменяется
```



Советуем В C++ это единственный способ передачи массивов как параметров. Чтобы различать входные и выходные массивы, используйте для входных массивов, не изменяемых функцией, модификатор const.

Передача массивов как параметров — эффективный механизм C++. Она не предполагает копирования данных, экономит память в стеке и время выполнения. Подобно передаче параметров-структур, отсутствие модификатора const должно свидетельствовать, что компоненты массива изменяются в функции. Это будет ваш вклад в борьбу с кризисом ПО.

Еще о преобразовании типов

Как уже упоминалось в разделе по преобразованию типов, в C++ реализован строгий подход: если в параметре ожидается скалярное значение, то структура или массив в качестве фактического аргумента использоваться не могут.

В отношении структур это правило расширено: если структура (или класс) представляет конкретный тип, ожидаемый в параметре, то скалярное значение, массив или структура другого типа в фактическом аргументе использоваться не могут — возникнет синтаксическая ошибка. Даже если структура другого типа имеет в точности тот же состав, что и ожидаемый тип, это не поможет. Когда поля обоих типов имеют одинаковый порядок, одни и те же типы и имена, компилятор все равно ждет аргумента именно того типа, имя которого совпадает с именем типа формального параметра, и больше ничего не анализирует.

Все сказанное относится к передаче структур по значению. Передача структур по указателю и по ссылке несколько отличается от передачи массивов.

В разделе по передаче параметров-структур обсуждались тип Account и функция swapAccounts(), в которой параметр Account передавался по указателю.

```
struct Account {
  long num;                                  // для упрощения - просто два поля
  double bal; } ;

void swapAccounts (Account *a1, Account *a2)          // нужен Account
{ Account temp;
  if ((*a1).num > (*a2).num)
  { temp = *a1; *a1 = *a2; *a2 = temp; } }
```

Теперь рассмотрим еще один тип — Transaction — и попытаемся передать его функции swapAccounts() переменную этого типа. Компилятор пресекает эту попытку, фиксируя синтаксическую ошибку:

```
struct Transaction {
  long num;                          // то же имя и тот же тип, что в Account
  double amt; } ;                    // другое имя, но такой же тип

Transaction tran1, tran2; ...        // код клиента
swapAccounts(&tran1,&tran2);         // ошибка: неверный тип аргумента
```

Но ведь это, по существу, одинаковые структуры, и хотелось бы использовать для них одну функцию swapAccounts(), а не писать еще одну — swapTransactions(). Я знаю, что делаю, и хотел бы убедить компилятор принять этот код, а не генерировать ошибку. C++ предоставляет такую возможность: явное приведение типа. Нужно лишь преобразовать указатель Transaction к типу указателя Account, после чего компилятор примет данный код:

```
swapAccounts((Account*)&tran1,(Account)&tran2);          // нет ошибки
```

Выглядит довольно неприглядно, но работает, подтверждая мое право на свободу действий. Другие программисты могут сказать, что я ищу приключений на свою голову. Действительно, в процессе сопровождения программы любой из типов — Account или Transaction — могут измениться, и тогда... Да поможет нам Бог. Возможно, надежнее будет все же написать небольшую функцию swapTransactions().

То же правило действует для передачи структур по ссылке. В листинге 7.5 уже показывалась функция printAccounts(), которая ожидает аргументы Account, переданные по ссылке. Если переменные Transaction передаются как фактические аргументы, то компилятор справедливо заметит, что данный код содержит синтаксическую ошибку:

```
Transaction tran1, tran2; ...                  // объекты транзакций
printAccounts(tran1,tran2);                    // синтаксическая ошибка:
                                               // неверный тип аргументов
```

Если бы я настаивал на своем праве сделать это, то мог бы использовать приведение типа Transaction к Account.

```
printAccounts((Account&)tran1,(Account&)tran2);
                                               // нет синтаксической ошибки
```

Это не очень хорошо с точки зрения разработки ПО, но C++ такое допускает. Обратите внимание, что приведение типа сообщает программисту, сопровождающему ПО, что я имел в виду при проектировании программного кода (использовал существующую функцию для другой цели вместо написания новой).

Аналогичная ситуация возникает в случае массивов, так как имя массива без модификаторов эквивалентно указателю на первый элемент массива.

Если бы функция использовала в качестве формального параметра массив конкретного типа, а вместо него подставлялась бы скалярная переменная, структурная переменная или массив, то возникла бы синтаксическая ошибка. C++ блюдет свою репутацию как язык со строгим контролем типов.

Рассмотрим, к примеру, функцию copyAccounts(), копирующую один массив типа Account в другой массив:

```
void copyAccounts(Account dest[], const Account src[], int size)
  { for (int i=0; i < size; i++)
    dest[i] = src[i]; }                          // тот же код, что и для Copy()
```

Если бы я попытался использовать данную функцию для копирования массива объектов Transaction или массива целых значений, то компилятор справедливо выразил бы свое возмущение:

```
Transaction tran1[5], tran2[5]; ...              // массивы транзакций
int data1[20], data2[20]; ...                    // массивы целых
copyAccounts(tran1,tran2,5);                     // синтаксическая ошибка:
                                                 // неверный тип аргумента
copyAccounts(data1,data2,20);                    // синтаксическая ошибка:
                                                 // неверный тип аргумента
```

Используя кастинг, можно заставить компилятор принять следующий вызов функций.

```
copyAccounts((Account*)tran1,(Account*)tran2,5);          // нет ошибки
copyAccounts((Account*)data1,(Account*)data2,20);         // нет ошибки?
```

Поскольку переменные типа Transaction имеют тот же размер, что и объекты типа Account, первый вызов функции может дать что-то разумное, в отличие от второго, где вместо int копируется 20 фрагментов памяти размером Account, что приводит к порче содержимого памяти.

То же самое относится к передаче массивов встроенных скалярных типов. Функция Copy() в предыдущем разделе требует в качестве аргументов массивов double. Если вместо них ей подсунуть int, компилятор выведет сообщение об ошибке. Если привести фактические аргументы к типу (double*), компилятор сгенерирует объектный код, копирующий память фрагментами длиной double, а не int.

К счастью, невозможно попасть в подобные ситуации по ошибке: чтобы компилятор принял такой код, нужно явное приведение типа.

Возврат значения из функции

Во всех предыдущих примерах функция возвращала тип void или встроенный скалярный тип int. Конечно, функции C++ возвращают данные по значению. Следовательно, копия возвращаемого значения в области действия функции присваивается переменной в области действия вызывающей программы.

C++ позволяет возвращать из функции структуру, если функция — структурного типа. В приведенных ниже примерах используется модифицированная версия функции swapAccounts(). Как и в предыдущей версии, она сравнивает номера счетов, заданных двумя аргументами Account, и меняет аргументы местами, если они не упорядочены (т. е. номер счета в первом аргументе больше, чем номер счета во втором). В отличие от предыдущей версии, функция возвращает переменную Account с большим номером счета (параметр a2).

```
Account swapAccounts (Account &a1, Account &a2)    // новый возвращаемый тип
{ Account temp = a1;
  if (a1.num > a2.num)
  { a1 = a2; a2 = temp; }
  return a2.num; }     // плохой возвращаемый тип: нет преобразования из long
```

Здесь применяются правила строгого контроля типов. Если возвращаемый тип определяется как структура (в данном случае Account), то такой же тип должен использоваться в двух других местах — в возвращаемом выражении (в функции) и в переменной в области действия вызывающей программы. Ни выражение, ни переменная в вызывающей программе не могут быть встроенного типа, иметь другой структурный тип или массив. Преобразование между этими типами невозможно:

```
Account ac1, ac2, ac3; long acc_num; ...        // значение в области действия
                                                // вызывающей программы
acc_num = swapAccounts(ac1,ac2);                // ошибка: нет преобразования
```

Аналогично передаче параметров возвращаемые из функции значения требуют координации кода в трех местах:

1) в возвращаемом типе;

2) в возвращаемом выражении;

3) в переменной в области действия вызывающей программы.
Вполне законно применение во всех этих трех местах
одного и того же типа:

```
Account swapAccounts (Account &a1, Account &a2)
{ Account temp = a1;                // инициализация temp значением a1
  if (a1.num > a2.num)              // проверка порядка счетов
  { a1 = a2; a2 = temp; }           // поменять местами, если аргументы
                                    // не упорядочены
  return a2.}                       // корректный тип возвращаемого выражения

ac3 = swapAccounts(ac1,ac2);        // корректный тип возвращаемого значения
```

Это допустимо, но при больших структурах и при частом вызове функции будет работать медленно. Именно поэтому большинство функций возвращают тип void, целое или булево значение, которые указывают на успешное или неуспешное выполнение функции.

C++ не позволяет функции возвращать массив, но предоставляет возможность возвращать указатель или ссылку. Это можно использовать для преодоления проблемы копируемого возвращаемого значения. В следующем примере функция swapAccounts() сравнивает поля num аргументов Account, меняет их местами, если нарушен порядок, и возвращает указатель на переменную Account, у которой номер счета больше:

```
Account* swapAccounts (Account &a1, Account &a2)    // возвращает указатель
{ Account temp = a1;
  if (a1.num > a2.num)
  { a1 = a2; a2 = temp; }
  return &a2; }                    // возвращает адрес фактического аргумента
```

Здесь типы также должны быть совместимы. Это касается трех мест:

1) объявленного типа, возвращаемого функцией;

2) типа выражения, возвращаемого функцией;

3) типа переменной в области действия.

```
Account ac1, ac2, ac3, *ac4; ...
ac4 = swapAccounts(ac1,ac2);          // ac4 - указатель, а не Account
ac4->num = 0;                         // влияет на неупомянутые ac1 или ac2
*ac4 = ac3;          // копирование ac3 в структуру с большим номером счета
```

Как видно, возврат указателя позволяет применять весьма туманные способы программирования в клиентском коде типа ac4->num = 0. Это может давать ссылку на ac1 или ac2 — при сопровождении приложения придется изучать реализацию функции-сервера, например swapAccount(). Если эта функция также написана не очень понятно, понадобится обращаться к другим сегментам кода. Все это усложняет задачи сопровождения и повышает вероятность ошибок. Возврат указателей дает возможность применять в клиенте более изощренный синтаксис. Например, можно присвоить 0 счету с большим номером при помощи строки:

```
swapAccount(ac1,ac2)->num = 0;          // неплохо, не правда ли?
```

Если нужно скопировать переменную ac3 в структуру с большим значением остатка на счете, то можно воспользоваться следующим оператором:

```
*swapAccount(ac1,ac2) = ac3;            // на самом деле это не очень хорошо
```

Такой код корректен, но не очень четко демонстрирует намерения разработчика. Приходится тратить лишние усилия на то, чтобы уяснить смысл. Если такой стиль программирования нежелателен, то разработчик функции swapAccount() может выразить то же самое, возвращая значение указателя на объект const.

```
const Account swapAccounts (Account &a1, Account &a2)          // новая идея
{ Account temp = a1;
  if (a1.num > a2.num)
  { a1 = a2; a2 = temp; }
  return &a2; }                    // возврат адреса фактического аргумента
```

Это означает, что возвращаемый адрес не может использоваться для изменения значения переменной, на которую он указывает. Если функция swapAccounts() определяется следующим образом, то такой код даст синтаксическую ошибку:

```
*swapAccounts(ac1,ac2)=ac3;             // ошибка: изменения в объекте const
                                        // не допускаются
swapAccounts(ac1,ac2)->num = 0;         // ошибка: нельзя изменять const
```

Применение подобного возвращаемого значения достаточно ограничено. Так как указатель невозможно использовать для изменения значения, на которое он ссылается, его нельзя присвоить произвольному указателю корректного типа.

```
Account *ac5 = swapAccounts(ac1,ac2);       // синтаксическая ошибка поэтому
ac5->num = 0;                               // данный код не компилируется
```

Такое возвращаемое значение можно применять только для доступа к компонентам объекта или присвоить его указателю на объект const:

```
const Account *ac5 = swapAccounts(ac1,ac2);     // теперь OK
ac5->num = 0;                                   // все равно не компилируется
```

Следует внимательнее обращаться с указателем, ссылающимся на объект в области действия вызывающей программы, который продолжает существовать после завершения серверной функции. Поэтому лучше не возвращать указатель на переменную, определяемую только в области действия функции-сервера. В предыдущем примере аккуратно возвращается указатель на параметр функции, представляющий собой указатель на фактический аргумент. Он продолжает существовать в области действия клиента после вызова. Следовательно, возвращаемый функцией адрес остается действительным. Однако это не всегда так.

Рассмотрим, например, следующую реализацию функции swapAccount(), возвращающую указатель на структуру. Номер счета для этой структуры берется из параметра a1:

```
Account* swapAccounts (Account &a1, Account &a2)     // возвращает указатель
{ Account temp = a1;          // temp содержит данные из a1
  if (a1.num > a2.num)
  { a1=a2; a2=temp; }         // a1 может измениться, но temp содержит его данные
  return &temp; }             // так чей это адрес?
```

Когда достигается закрывающая фигурная скобка области действия функции, переменная temp уничтожается. Следовательно, указатель ac4 в клиентском коде ссылается не на адрес структуры, содержащей данные из переменной ac1, а на память, уже не принадлежащую программе. Это называется "повисшим" указателем. Такой указатель ссылается на уже исчезнувший объект.

Не каждая среда выполнения программ обладает средствами для перехвата подобного нарушения работы с памятью, но в некоторых они есть. Более того, память, занятая для temp, в течение некоторого времени не может использоваться для других целей, и клиентский код с таким адресом даст корректные результаты.

Вновь и вновь мы сталкиваемся с ситуацией, когда приемлемость исходного кода для компилятора и корректность результатов на этапе выполнения при прохождении всех ветвей программы не может быть достаточным основанием для корректности программы в целом.

Возвращать указатели на локальные переменные небезопасно. Более надежно возвращать указатели на переменные в динамически распределяемой области памяти или на переменные в области действия клиента. Вот пример возврата указателя на переменные в области действия клиента, когда решается проблема "повисшего" указателя:

```
Account* swapAccounts (Account &a1, Account &a2)     // возвращает указатель
{ Account temp = a1;                                 // temp содержит данные из a1
  if (a1.num > a2.num)
  { a1=a2; a2=temp;
    return &a2; }                                    // данные из a1 теперь в a2
  return &a1; }                                      // данные из a1 теперь снова в a1
```

При возврате указателей на переменные в области действия клиента следует знать, являются ли эти переменные константами. Рассмотрим пример функции, сравнивающей поля bal двух переменных типа Account и возвращающей указатель на объект с большим остатком на счете.

```
Account* largerBalance (const Account &a1, const Account &a2)      // нет!
{ return (a1.bal>a2.bal) ? &a1 : &a2l }   // указатель на фактический аргумент
```

Это хороший пример работы с объектами-константами. Функция не изменяет состояния параметров, а только обращается к ним как к входным значениям для вычислений. Именно поэтому в заголовке функции используется модификатор const. Между тем данная функция не компилируется, потому что она возвращает указатель, не определенный как указатель на объект-константу. Данная функция обещает не изменять состояния фактических аргументов, но возвращает указатель, ссылающийся на один из этих аргументов, следовательно, может модифицировать его поля. Пометив такой код как синтаксическую ошибку, компилятор предотвращает создание клиентского кода подобного типа, где модифицируется объект-константа.

```
const Account acc1 = {325,1000.0}, acc2 = {370,100.0};       // не изменяется
Account *p = largerBalace(acc1,acc2);      // допустимый, но опасный синтаксис
p->bal = 0;          // допустимый синтаксис, но модифицирует объект-константу
```

Это кажется избыточной мерой. Хотя параметры функции largerBalance() определяются как константы, они могут передавать отличные отconst аргументы:

```
Account acc1 = {325,1000.0}; acc2 = {370,100.0};          // объект изменяется
Account *p = largerBalace(acc1,acc2);      // допустимый, но опасный синтаксис
p->bal = 0;          // OK для не константы, но не годится для объекта-константы
```

Однако компилятор не настолько интеллектуален, чтобы различать функции между передаваемыми объектами-константами и не константами. Как иногда бывает в реальной жизни, решение состоит в том, чтобы просто забыть обо всем остальном, поэтому C++ требует указывать в возвращаемом типе модификатор const (обратите внимание, что я не предлагаю убратьconst из заголовка функции)

```
const Account* largerBalance (const Account &a1, const Account &a2)
{ return (a1.bal>a2.bal) ? &a1 : &a2; }            // такой код компилируется
```

Такой вариант компилируется нормально, однако каким образом это предотвращает изменение объекта-константы? Очень просто. Следующий фрагмент компилироваться не будет:

```
const Account acc1 = {325,1000.0}, acc2 = {370,100.0};       // не изменяется
Account *p = largerBalance(acc1,acc2);     // теперь это синтаксическая ошибка
p->bal = 0;                                // нет синтаксической ошибки, но
                                           // компилятор такого не разрешает
```

Компилятор хочет предотвратить присваивание вида p->bal = 0: операция синтаксически корректна, но он помечает присваивание указателю p, поскольку это не гарантирует от модификации указываемого объекта. Компилятор заставляет программиста координировать свои действия и определить p как указатель на объект-константу.

```
const Account acc1 = {325,1000.0}, acc2 = {370,100.0};       // не изменяется
const Account *p = largerBalance(acc1,acc2);                 // OK, нет ошибки
p->bal = 0;                                 // нет синтаксической ошибки, но
                                            // компилятор такого не разрешает
```

Когда объекты Account не определяются как неизменяемые, это будет избыточным — указатель p нельзя использовать для их модификации. Данная избыточность — плата за безопасность объектов-констант (и за нежелание разработчиков компилятора анализировать поток данных в программе).

Самый безопасный способ использования адресов для возврата значений — это динамическое управление памятью. Функция-сервер распределяет динамическую память и возвращает указатель на эту память клиенту. (Некоторые другие функции должны позднее освобождать данную память.) В нашем примере функция allocateAccounts() распределяет динамический массив объектов Account. Размер массива передается как аргумент.

```
Account* allocateAccounts(int size)        // указатель на не константу
  { if (size <= 0) return 0;               // проверка допустимости аргумента
    Account *p = new Account[size];
    if (p == 0)                            // просто, но слишком "сыро"
       cout << "В allocateAccounts() нет памяти\n";
    return p; }                            // NULL, если что-то не так
```

Если что-то не так, функция возвращает указатель NULL. Клиент должен проверять, успешно ли выделена память.

Возврат указателя — еще одна альтернатива возврату значения структурной переменной, позволяющая избежать копирования значений на этапе выполнения.

Концептуально она аналогична возврату указателя на структуру, но с практической точки зрения весьма отличается, поскольку в C++ по умолчанию ссылки являются константами. Рассмотрим версию функции swapAccounts(), которая возвращает ссылку на фактический аргумент с большим номером счета.

```
Account& swapAccounts (Account &a1, Account &a2)
{ Account temp = a1;
  if (a1.num > a2.num)
  { a1=a2; a2=temp; }
  return a2; }                    // неверный тип, если возвращается &a2
```

Обратите внимание, что эта версия аналогична версии, возвращающей структуру по значению. Единственная разница в операции & в возвращаемом типе. Было бы некорректно возвращать &a2 вместо a2, так как a2 имеет тип Account и может использоваться для инициализации ссылки типа Account, а &a2 — указатель типа Account и не может применяться для инициализации ссылки Account. Эти два типа в C++ несовместимы.

При этом в клиентском коде могут легко возникнуть проблемы:

```
Account ac1,ac2,a3, &ac4; ...          // здесь это ссылка
ac4 = swapAcccounts(ac1,ac2);          // фантазии, а не реальный код
```

Этот фрагмент некорректен. Здесь возвращаемое функцией swapAccounts() значение присваивается переменной ac4, но слишком поздно. Ссылки должны инициализироваться при определении, а выше этого не сделано. Единственный способ присвоить такое возвращаемое значение — использовать его для инициализации:

```
Account ac1,ac2,a3;...
Account &ac4 = swapAcccounts(ac1,ac2);              // на этот раз OK
ac4.num = 0;            // влияет на не упомянутые здесь ac1 или ac2
ac4 = a3;               // копирование a3 в структуру с большим номером счета
```

Поскольку ac4 — синоним ac1 или ac2, результат данного кода не вполне ясен. Так как возвращаемое функцией swapAccounts() значение представляет собой ссылку, можно использовать для возврата указателя вот такой хитрый синтаксис:

```
largerBalance(ac1,ac2).num = 0;          // прекрасно, не правда ли?
largerBalance(ac1,ac2) = a3;             // это тоже неплохо
```

Во всех языках, включая язык C, такого рода синтаксис запрещен. Возвращаемое функцией значение нельзя использовать как l-значение. Но C++ это разрешает, однако алгоритм все равно лучше составлять так, чтобы не было необходимости в подобных вычислениях.

В предыдущем примере переменная ac4 была благоразумно определена как ссылка, а не как структурная переменная. Если бы для сохранения значения, возвращаемого функцией по ссылке, использовалась структурная переменная, копирование выполнялось бы точно так же, как при возврате по значению:

```
ac3 = largerBalance(ac1,ac2);       // ac3 не ссылка, а переменная типа Account
```

Поскольку ac3 представляет собой оригинальный объект структурного типа, все преимущества возврата значения по ссылке теряются.

Данная тема достаточно сложна и требует учета множества других моментов. Чтение и понимание программного кода, где возвращаются структурные значения, указатели и ссылки — задача нетривиальная. Стоит ли прибегать к этому ради удобства или производительности? Существуют ли другие, более простые способы получения того же результата?

Возможно, неплохо было бы возвращать из функций C++ только логические флаги, показывающие успешное или неудачное выполнение функции. Иногда (особенно при динамическом распределении памяти) имеет смысл и возврат указателя. Каждый раз, когда возвращается указатель или ссылка, нужно убедиться, что это действительно дает преимущества в производительности и не нарушает целостности программы.

Встраиваемые функции

Еще одна полезная техника C++, способствующая модульности программы, это так называемые встраиваемые (или подставляемые) функции C++. Как уже говорилось выше, копирование аргументов и переключение контекста при вызове функции может повлиять на размер стека программы и ее производительность. Это очень важные вопросы. Если функция небольшая и часто вызывается из функций с большим числом локальных переменных, то жалко тратить время и место в стеке на сохранение среды вызывающей программы ради выполнения нескольких строк кода.

Рассмотрим, например, функцию, вычисляющую размер налога с помощью постоянного коэффициента.

```
double tax(double gross)
{ return gross * 0.05; }
```

Когда клиентский код вызывает эту функцию, "контекст" функции-клиента (ее параметры и локальные переменные, включая массивы) сохраняются в стеке. Когда функция завершает работу, контекст восстанавливается:

```
double sales, state; ...
state = tax(sales);                    // вызов функции
```

Было бы неплохо избежать этих непроизводительных издержек при вызове такой маленькой функции. В языке C эта проблема решается с помощью макрокоманд — литеральной подстановки и имитации вызова функции:

```
#define tax(x) x*0.05
```

Клиентский код `state = tax(sales);` расширяется в `state = sales*0.05;` — так удается избежать непроизводительных потерь при вызове функции.

C++ поддерживает такие же возможности макрокоманд, как и язык C, однако расширение макрокоманд выполняется препроцессором, а не транслируется компилятором. Макрокоманда — не функция. Она не может иметь локальных переменных, не предусматривает проверки типа параметров и невидима для отладчика.

Кроме того, достаточно трудно работать с макрокомандами, занимающими несколько строк исходного кода. Когда в расширяемом коде содержатся синтаксические ошибки, компилятор сообщает номер строки исходного кода, вызывающего макрокоманду, а не той строки, где эта макрокоманда определена. Если макрокоманда содержит несколько строк, то довольно трудно понять, какая из них привела к сообщению об ошибке.

Макрокоманды ничего не знают о приоритете выполнения операций в C++. Они просто служат для литеральной подстановки текста безотносительно к реальной задаче. Рассмотрим, например, следующий клиентский код:

```
state = tax(sales+20.2);               // выражение как фактический аргумент
```

Для программиста это означает:

```
state = (sales+20.2) * 0.05;           // желательная интерпретация
```

Но препроцессор вычисляет выражение, используя литеральную подстановку макрокода:

```
state = sales+20.2 * 0.05;                    // интерпретация препроцессора
```

Конечно, с данной проблемой можно справиться (например, включив в макроопределение скобки), но пример демонстрирует, что макрокомандам свойственны недостатки, которых лучше избегать. C++ позволяет программисту объявить функцию как встраиваемую (подставляемую). Это средство подобно макрокоманде, но не имеет недостатков оператора препроцессора `#define`.

Если функция содержит модификатор `inline`, любое обращение к ней заменяется операторами, которые находятся в ее определении. Никакого вызова функции не происходит, а следовательно, не используется пространство в стеке:

```
inline double tax(double gross)
{ return gross * 0.05; }
```

В то же время функция `inline` — действительно функция. Она может содержать несколько строк, определять вложенные блоки и иметь локальные переменные. Как и любая функция C++, функция `inline` допускает контроль типов параметров и операции отладки.

Данное средство позволяет использовать все преимущества модульного программирования без непроизводительных потерь на переключение контекста (в начале и в конце вызова функции). Тело встраиваемой функции включается в клиентский код при каждом вызове. В исходном коде клиента будет столько копий функции `inline`, сколько раз она вызывается.

Встраиваемые функции повышают производительность программы, но, если эти функции не вносят весомую долю в общее время выполнения программы, выигрыш может быть невелик. Вместе с тем из-за встраиваемых функций растет объем выполняемого файла программы, что может увеличить обмен с диском (свопинг) и даже уменьшить скорость выполнения.

Модификатор `inline` не является для компилятора безусловной командой. Это лишь предложение. Если функция слишком сложная или слишком длинная, компилятор может игнорировать его (таково мнение разработчиков компилятора).

Некоторые компиляторы не воспринимают в функциях `inline` никаких управляющих конструкций. Другие допускают один или два оператора `if`, но игнорируют функции, содержащие циклы. Используйте данное средство только для простых функций.

Определение большого числа функций как встраиваемых не повышает производительности программы, поэтому используйте модификатор `inline` только для тех функций, вызовы которых действительно влияют на производительность. Если необходимо, профилирование программы поможет идентифицировать "узкие" места.

В главе 2 уже упоминались два способа определения функций — компонентов класса (структуры). Один из них состоит в реализации этих функций в границах спецификации класса. Другой — в использовании в спецификации класса прототипов функции и реализации самих функций в другом месте. Компонентные функции, определяемые в спецификации класса, являются встраиваемыми по умолчанию (неявно). Модификатор `inline` для них не нужен:

```
struct Counter {
private:
    int cnt;
 public:
    void InitCnt(int Value)
    { cnt = Value; }                  // inline по умолчанию
    void UpCnt()
    { cnt ++; }
```

```
    void DnCnt()
    { cnt-; }
    int GetCnt()
    { return cnt; } } ;
```

В спецификации класса обычно задаются только прототипы функций, а не их реализация.

```
struct Counter {
private:
    int cnt;
  public:
    void InitCnt(int);              // только прототип
    void UpCnt();
    void DnCnt();
    int GetCnt(); } ;
```

Если компонентная функция реализована вне фигурных скобок класса, то она не является по умолчанию встраиваемой. Ее можно определить как встраиваемую с помощью ключевого слова inline:

```
void Counter::InitCnt(int Value)
  { cnt = Value; }
inline void Counter::UpCnt()
  { cnt++; }
inline void Counter::DnCnt()
  { cnt-; }
inline int Counter::GetCnt()
  { return cnt; }
```

О классах подробнее рассказывается в следующих главах.

Параметры с заданными по умолчанию значениями

Это новое средство языка, недоступное в языке C. Оно введено с целью дальнейшего улучшения читабельности кода и удобства его модификации. При объявлении функции можно задать значения по умолчанию для одного или нескольких параметров в списке.

Ниже приведено объявление функции sum(), вычисляющей сумму заданного числа компонентов массива значений double. В этом объявлении используется синтаксис инициализации второго формального параметра. Для него по умолчанию задается значение 25:

```
double sum (const double a[], int n=25);        // прототип
```

Этот синтаксис инициализации говорит компилятору, что при отсутствии фактического значения при вызове в клиентском коде нужно использовать значение, заданное в прототипе.

```
double total; double x[100]; int n; ...        // и т. д.
total = sum(x);                    // складывает 25 компонентов массива x[]
```

Конечно, клиентский код может переопределить значение по умолчанию путем явного задания значения фактического аргумента:

```
total = sum(x,n);                  // складывает n компонентов массива
```

На первый взгляд, это мало что дает. Если нужно сложить 25 компонентов массива, почему бы не сделать это явно? На самом деле, применение используемых по умолчанию значений параметров усложняет программный код, но в некоторых случаях, когда функция содержит большое число аргументов и вызывается часто лишь с некоторыми аргументами, а другие включаются в вызов редко, может упростить его. Например, функция `getline()` в файле `istream.h` имеет следующий прототип:

```
istream& getline(char buf[], int count, char delimiter = '\n' );
```

Обычно она вызывается только с двумя параметрами: символьным массивом для чтения в него данных и максимальным числом сохраняемых символов (включая нулевой завершающий символ), если ранее в строке ввода не встречается символ новой строки. Эта функция позволяет также использовать произвольные ограничители, такие, как знак доллара, фунта и др. В данном случае применение значений по умолчанию освобождает программиста от необходимости каждый раз указывать стандартный ограничитель '\n' (если в качестве ограничителя используется символ новой строки).

Обратите внимание, что значения по умолчанию для параметров следует задавать в прототипах, а не в определении функции. Определение функции в таком случае не может содержать значение по умолчанию:

```
double sum(const double a[],int n)          // значение по умолчанию
                                            // не используется
 { double total = 0.0;
   for (int i=0; i<n; i++)
      total += a[i];
   return total; }
```

Это означает, что разработчик функции может даже не знать, что в клиентском коде используются значения по умолчанию. Разные функции, реализуемые в разных файлах, могут объявлять прототипы с разными значениями по умолчанию для одного и того же параметра. Координировать их использование не требуется.

В одном файле можно указывать для параметра только одно значение по умолчанию. Если функция определяется в том же файле, где она используется, то, если в тот же файл не включается прототип функции, значение по умолчанию можно указывать в определении функции. Значит, когда файл с функцией содержит и прототип, и определение функции, значение по умолчанию указывается только в одном месте. Аналогично, если в один файл включаются два прототипа функции, то значение по умолчанию можно определять лишь в одном из них. Если значение по умолчанию задается в двух прототипах, это будет синтаксическая ошибка, даже когда это одно и то же значение. Компилятор не сравнивает значения по умолчанию, а сразу обвиняет программиста в переопределении данного значения.

Поскольку имена параметров в прототипах функции не обязательны, можно "инициализировать" значением по умолчанию имя типа, а не имя параметра.

```
double sum (const double a[], int=25);          // присвоить типу int?
```

Неужели здесь действительно присваивается 25 типу `int`? Конечно, нет. Это вовсе не присваивание, а просто обозначение, цель которого — сообщить компилятору (и программисту, сопровождающему приложение) о существовании значения по умолчанию.

Довольно типичное решение разработчиков языка C++. В нем значительно расширены возможности языка C. Язык разросся, в нем появилось много новых ключевых слов и операций. Между тем количество символов операций ограничено, и уже в языке C использовались некоторые двухсимвольные операции. Так как число разумных комбинаций символов не очень велико, в C++ добавлены также ключевые слова, но необходимость изучения большого числа ключевых

слов создает впечатление перегруженности языка. C++ является надстройкой языка C и пытается сохранить репутацию компактного языка, простого в изучении и применении. Именно поэтому в C++ новые ключевые слова добавлены лишь для необходимости (хорошие примеры: new,delete, class, public, private и protected). По этой же причине C++ допускает повторное использование операций и ключевых слов для других целей. Мы уже видели, что операцию получения адреса & можно применять как операцию ссылки. В прототипе функции sum() C++ использует для новой цели операцию присваивания — с ее помощью задается значение параметра по умолчанию.

На самом деле это неплохая стратегия разработки языка. Она уменьшает число символов и ключевых языков, которые приходится изучать и осваивать программисту. Но для каждой повторно используемой операции нужно изучать разные случаи применения, а это может запутать программиста. Один из таких примеров — повторное использование операции &, имеющей разный смысл в разных контекстах. Неопытных программистов это иногда смущает.

C++ допускает применение значений по умолчанию только для самых правых параметров. В середине списка параметров значения по умолчанию указывать нельзя.

```
int foo(int a=0,int b=2,double d1,double d=1.2);          // нельзя
```

Здесь либо нужно убрать левые значения по умолчанию (для обоих параметров int), либо дать значение по умолчанию первому параметру double.

Это нельзя считать серьезным ограничением. Ведь в любом случае значение по умолчанию можно явно переопределить.

Использование операции присваивания как операции для значения по умолчанию может привести к проблемам, если его спутать с обычной операцией присваивания. Рассмотрим, например, функцию, динамически создающую новый узел (типа Node), инициализирующую его поле (типа Item) и ссылку на следующий узел в связанной структуре (типа Node*):

```
Node* createNode(Item item, Node* next)
{ Node *p = new Node;            // распределение памяти в динамической области
  p->item = item; p->next = next;     // инициализация полей узла
  return p; }                         // указатель для использования в клиенте
```

Во многих приложениях новый узел добавляется к концу связанного списка, а его следующее поле устанавливается в значение NULL, указывая, что это последний узел в списке. Следовательно, клиент может вызвать функцию createNode() со значением 0 (или NULL) во втором аргументе:

```
tail->next=createNode(item,0);          // добавление узла к концу списка
tail = tail->next;                      // указывает на новый последний узел
```

Возлагать на клиента обязанности спецификации нулевого значения при каждом использовании функции createNode() как сервера неправильно. Эти обязанности следует возложить на сервер. Тогда программный код клиента будет выглядеть так:

```
tail->next=createNode(item);            // добавление узла к концу списка
tail = tail->next;                      // указывает на новый последний узел
```

Одно из возможных решений данной проблемы — применение значения по умолчанию:

```
Node* createNode(Item item, Node* next=0);       // прототип
```

Между тем, пропуск имен параметров в прототипе неожиданно создает новую проблему:

```
Node* createNode(Item, Node*=0);              // что это означает?
```

Это синтаксическая ошибка. Компилятор возмущается, так как думает, что здесь используется операция *=. Хотя он не прав, оправдаться трудно. Единственный способ умилостивить его — добавить пробел между звездочкой и символом равенства.

```
Node* createNode(Item, Node* =0);          // это лучше
```

Вы помните о том, что, подобно языку C, язык C++ равнодушен к пробелам? Да, в C это действительно так, а в C++ — за некоторыми исключениями. Исключения как раз и вызваны необходимостью использовать операции в разных целях.

Параметры по умолчанию полезны в тех приложениях, где некоторые функции часто вызываются с одними и теми же значениями переменных, соответствующими контексту применения. Если конкретные значения параметров используются только в конкретных обстоятельствах, то применение значений по умолчанию оправдано. Это типично, например, для программирования в Windows.

Неразборчивое использование значений по умолчанию затрудняет понимание исходного кода клиента, и его следует избегать.

В некоторых случаях значения параметров по умолчанию могут помочь совершенствованию программы и добавлению нового кода вместо изменения существующего.

Рассмотрим простую функцию registerEvent(), применяемую в системе реального времени:

```
inline void registerEvent()
{ count++; span = 20; }                    // увеличить счетчик событий,
                                           // установить интервал времени
```

Функция производит не только это, но здесь специально убраны лишние детали, не относящиеся к делу. Оставлен только счетчик событий и задание интервала времени с помощью глобальных переменных. Стоит отметить, что это сложная и большая система, а вызовы данной функции содержатся примерно на 400 страницах исходного кода.

```
registerEvent();                           // вызов сервера в клиентском коде
```

При дальнейшем развитии системы и ее сопровождении случается неизбежное. Система должна работать с другими видами событий, а для этих событий нужно отдельно устанавливать интервал времени. 400 страниц исходного кода не требуют изменений, ведь основное событие обрабатывается точно так же, как раньше, но примерно на 10 страницах приходится иметь дело и с основным событием, и с новым.

Один из способов справиться с такого рода проблемой состоит в написании новой функции, например regEvent():

```
inline void regEvent(int duration)         // еще одна функция-сервер
{ count++; span = duration; }              // увеличение счетчика событий
```

Вполне жизнеспособное решение, но чередование вызовов функций registerEvent() и regEvent() может только запутать. Кроме того, потребуется новое имя функции, а это всегда усложняет сопровождение. Наконец, лучше, когда аналогичные действия выполняются с помощью одной и той же функции. Если нужно нарисовать фигуру или установить контекст для ее изображения, то лучше, если именами соответствующих функций будут draw() и setContext(), а не draw1() и setContext1() или что-то подобное.

Таким образом, хорошо бы изменить функцию registerEvent(), введя дополнительный параметр и модифицировав тело функции в соответствии с новыми требованиями:

```
inline void registerEvent(int duration)    // изменен заголовок
{ count++; span = duration; }              // тело тоже изменено
```

Теперь надо поменять вызов функции registerEvent() на 10 страницах кода с разными значениями фактического аргумента:

```
registerEvent(50); registerEvent(20);          // новый клиентский код
```

Кроме того, придется менять вызовы registerEvent() на 400 страницах исходного кода.

```
registerEvent(20);          // модифицированный вызов
                            // функции сервера в клиентском коде
```

Данное решение требует:

1. Добавления нового клиентского кода
2. Изменения заголовка существующей функции-сервера
3. Изменения тела существующей функции-сервера
4. Модификации имеющегося клиентского кода

Когда приходится координировать программный код в четырех местах, велика вероятность ошибки. Это особенно относится к данному случаю. Применение параметра с назначенным по умолчанию значением — хороший вариант. В этом случае придется изменить существующую функцию-сервер — модифицировать ее заголовок и тело:

```
inline void registerEvent(int duration)        // изменение заголовка
{ count++; span = duration; }                  // и тела функции тоже
```

Прототип функции в новом и существующем клиентском коде может выглядеть так:

```
inline void registerEvent(int duration=20);      // прототип
```

Тем самым устраняется необходимость наиболее обременительной деятельности в приведенном выше списке — модификации существующего кода в результате внесения изменения в другом месте (в данном случае — в функции-сервере). Возможно, это наиболее неприятная часть сопровождения программы. Вопрос даже не в трудозатратах, а в том, чтобы действительно внести все необходимые изменения там, где это требуется, и не изменить то, что менять не следует. Кроме того, чтобы убедиться в корректности изменений, потребуется регрессионное тестирование. Его трудно планировать и реализовывать и практически невозможно документировать.

Конечно, не все сопровождение можно свести к использованию значений по умолчанию, но нужно убедиться в том, что вы не упустили шанса применить их там, где это можно сделать. Они представляют серьезное усовершенствование по сравнению с традиционной технологией сопровождения ПО.

Перегрузка имен функций

Перегрузка (переопределение) имен функций — еще одно усовершенствование, способствующее модульному программированию на C++.

В большинстве языков каждое имя связывается с уникальным объектом в области действия (блоке, функции, классе, файле или программе). Это касается имен типов, переменных и функций.

В языке C для функций нет вложенных областей действия (функция в функции) и их имена должны быть уникальны в области действия программы, а не только в области действия файла. Два определения функции с одним и тем же именем в исходном файле приведут к ошибке при компоновке. В языке C не принимаются в расчет типы параметров или возвращаемых значений. Важно только имя функции, и оно должно быть уникально в проекте (включая библиотеки).

В C++ каждый класс имеет собственную, отдельную область действия. Следовательно, одно и то же имя может использоваться для функции-члена и для глобальной функции. Кроме того, одинаковые имена функций допускаются для функций-членов разных классов. Обратите внимание, что применение одного имени в разных областях действия не требует никаких различий в числе и типах параметров. Они могут быть одинаковыми или разными — это не важно. Если две функции определяются в разных областях действия (глобальной и области действия класса или в двух разных областях класса), то конфликта имен не будет.

Это действительно важное улучшение в технологии разработки ПО. Требование языка C состоит в том, чтобы все имена функций были уникальными, а это слишком серьезное ограничение, особенно для значительных по объему проектов. Быстрое увеличение количества имен затрудняет управление проектом. В крупных проектах координация между командами программистов, работающих над разными частями проекта, значительно усложняет дело. Введение классов в C++ устраняет большую часть подобных проблем. Но не все.

В C++ применимы те же правила области действия, что и в языке C. Вводимые программистами имена должны быть уникальными в той области действия, где они определены (это область действия класса или файла для имен типа и имен переменных, область действия проекта для имен функций). Было бы удобно использовать одно имя для разных функций в одной и той же области действия, а не только в разных.

C++ предлагает еще одно значительное усовершенствование в данной области — он допускает перегрузку имен функций. Смысл имени функции в C++ зависит от числа ее параметров и их типов. Использование одного имени для разных функций с разным числом параметров называется перегрузкой имен (overloading). Компилятор различает такие перегруженные функции.

Приведем пример применения одного имени функции `add()` для двух разных функций. У них различается число параметров. Одна функция имеет два параметра, другая — три:

```
int add(int x, int y)                // два параметра
{ return x + y; }

int add(int x, int y, inx z)         // три параметра
{ return x + y + z; }
```

Если список параметров для разных функций различен, компилятор C++ интерпретирует их как разные функции и одинаковое имя его не смущает. Когда такая функция вызывается клиентом, передаваемый клиентом список параметров заставляет компилятор выбрать правильное определение функции:

```
int a = 2, b = 3, c, d; ...          // и т. д.
c = add(a,b);                        // вызов int add(int x, int y);
d = add(a,b,c);                      // вызов int add(int x, int y, inx z);
```

Если число параметров у функций совпадают, но у них разный тип, перегрузка имен также возможна — ведь списки параметров будут отличаться друг от друга:

```
void add(int *x; int y)              // также два параметра
{ *x += y; }
```

Данная функция `add()` также имеет два параметра, но у первого параметра другой тип — "указатель на `int`", а не `int`. Этого достаточно, чтобы компилятор различал вызовы двух функций:

```
int a = 2, b = 3, c, d; ...          // и т. д.
c = add(a,b);                        // вызов int add(int x, int y);
d = add(a,b,c);                      // вызов int add(int x, int y, inx z);
add(&a,b);                           // вызов void add(int *x; int y)
```

Как видно, в C++ смысл вызова функции зависит от ее контекста — типов фактических аргументов, подставляемых в клиенте. Для разрешения неоднозначности компилятор C++ использует сигнатуру функции. Сигнатура — это просто общедоступный интерфейс функции. Он основывается на числе и типе ее аргументов. Разного порядка типов параметров достаточно, чтобы функции различались.

Конечно, имена аргументов при перегрузке имен функций в расчет не принимаются. Функции должны различаться по типам или по числу параметров.

```
double add(int x, int y)        // сигнатура та же: синтаксическая ошибка
{ double a = (double)x, b = (double)y;
  return a + b; }               // другой возвращаемый тип: этого недостаточно
```

Компилятор C++ не может отличить эту функцию от первой функции add(), возвращающей int:

```
int a = 2, b = 3, c, d; double e; ...      // и т. д.
c = add(a,b);                              // неоднозначность: какая функция?
e = add(a,b);                              // неоднозначность: какая функция?
```

Здесь первый вызов используется для присваивания значения целочисленной переменной, а второй — для присваивания переменной типа double. Читателю этого достаточно, чтобы понять, какая именно функция add() вызывается, но не компилятору.

Если две функции add() определены в разных файлах, но вызываются из одного клиентского кода, то компилятор помечает второй прототип как попытку переопределить уже определенную функцию:

```
int add(int x, int y);          // допустимый прототип
double add(int x, int y);       // переопределение функции:
                                // синтаксическая ошибка
```

Обратите внимание, что если возвращаемые функцией типы совпадают, компилятор принимает второй прототип за простое повторное объявление функции:

```
int add(int x, int y);          // допустимый прототип
int add(int x, int y);          // повторное объявление функции: нет проблем
```

Различия в коде функции также не учитываются. Программист сам должен обеспечить выполнение перегруженными функциями семантически подобных операций (как подразумевают их общие имена). Например, программист может написать еще одну функцию add() с четырьмя параметрами, возвращающую аргумент с максимальным значением:

```
int add(int a, int b, int c, int d)   // еще одна совмещенная функция add()
{ int x = a>b?a:b, y = c>d?c:d;       // плохое применение оператора условия
  return x>y ? x : y; }               // возвращает максимальное значение
```

Для компилятора C++ это вполне законно. Он будет отличать данную функцию от других функций add(), руководствуясь их интерфейсами. Что касается вашего начальника (и сопровождающего приложение программиста), то нетрудно догадаться об их мнении на этот счет.

Перегрузка имен устраняет необходимость придумывать уникальные имена для разных, но родственных функций:

```
int addPair (int, int);         // вместо int add(int x, int y);
int addThree(int,int,int);      // вместо int add(int,int,int);
void addTwo(int *, int);        // вместо void add(int*, int);
```

Ни компилятору, ни программисту не составит труда догадаться, какая именно функция вызывается клиентом.

Если компилятор не может сопоставить фактические аргументы ни одному набору формальных параметров для функции с указанным именем, он выводит сообщение о синтаксической ошибке. Если невозможно точное соответствие, то компилятор использует преобразование типов. В данном примере предполагается, что тип Item — это структура, совместимая с типом int:

```
int c; Item x; ...                  // и т. д.
c = add(5,x);                       // нет соответствия: синтаксическая ошибка
c = add(5,'a');                     // нет ошибки: приведение типа
c = add(5,20.0);                    // нет ошибки: преобразование
```

В применении символа там, где ожидается целое значение, смысла немного. Следовало бы запретить такие действия, но это не сделано, поэтому старайтесь не применять их, если на то нет весомых причин (трудно найти реальные причины, но... никогда не говори "никогда").



Внимание Если для класса определены операции или конструкторы преобразования, для аргументов-классов компилятор C++ применяет преобразования, заданные программистом (ниже об этом будет рассказано подробнее).

Когда две перегруженные функции имеют одно и то же число параметров, но у параметров разные типы, допускается преобразование типов, поэтому, чтобы избежать неоднозначности, лучше подставлять фактические аргументы точно соответствующих типов. Предположим, имеются две функции max(), одна из которых имеет параметры типа int, а другая — double:

```
long max(long x, long y)            // возвращает максимальное значение
{ return x>y ? x : y; }

double max(double x, double y)      // отличается от long
{ return x>y ? x : y; }
```

Когда типы аргументов в точности совпадают с формальными параметрами, компилятору C++ не стоит труда найти в клиенте верную функцию:

```
long a=2, b=3, c;
double x=2.0, y=3.0, z;
c = max(a,b);       // нет неоднозначности вызова: long max(long, long);
z = max(x,y);       // нет неоднозначности вызова: double max(double, double);
z = max(a,y);       // неоднозначность: какая функция
```

Здесь в последнем вызове функции первый фактический аргумент имеет тип long, а второй — double. Хотя возвращается значение типа double, компилятор отказывается различать, какая функция вызывается. Это можно указать явно, приведя аргумент к соответствующему типу:

```
z = max((double)x,y);           // нет неоднозначности
                                // вызова: double max(double, double);
```

В следующем примере делается попытка передать аргумент типа int. Очевидно, преобразование из int в long более естественно, чем из int в double, не так ли? Нет, не так. Возможно, это естественно для человека, но не для компилятора C++. В C++ нет такого понятия как сходство типов. Преобразование есть преобразование.

```
int k=2, m=3, n;
n = max(k,m);               // неоднозначность: какая функция? long? double?
```

Компилятору все равно, преобразовывать int в long или int в double, а потому он помечает такой вызов как неоднозначный.

Применение такого превосходного средства, как перегрузка имен функций — само по себе достаточно сложно. Возможно, применение двух функций — maxLong() и maxDouble() — не такая уж плохая идея. Особенно, если это еще не конец. Рассмотрим еще две перегруженные функции.

```
int min (int x, int y)                  // возвращает минимальное значение
{ return x>y ? x : y; }

double min(double x, double y)          // отличается от int
{ return x>y ? x : y; }
```

Давайте сыграем в ту же игру под названием "неоднозначность". Ответ вы знаете — компилятору все равно, преобразовывать int в long или int в double. Следовательно, такой вызов даст синтаксическую ошибку:

```
long k=2, m=3, n;
n = min(k,m);                   // неоднозначность: какая функция? int? double?
```

Рассмотрим то же самое для фактических аргументов типа short и float. Можно было бы ожидать от компилятора той же реакции, однако он транслирует этот исходный код без возражений:

```
long a=2, b=3, c;
float x=2.0f, y=3.0f, z;
c = min(a,b);               // нет неоднозначности вызова: int max(int, int);
z = min(x,y);               // нет неоднозначности вызова:
                            // double max(double, double);
```

Причина в том, что компилятор в этом случае не выполняет преобразования типов. Значения типа short не преобразуются в int, а просто приводятся к типу большего размера (promotion). Аналогично значения типа float приводятся к double. После такой операции компилятор может сопоставить типы аргументов и находит точное соответствие. Никакой неоднозначности нет.

Когда аргументы передаются по значению, спецификатор const считается излишним. Следовательно, перегруженные функции в этом случае различить невозможно. Например, такая функция не отличается от функции int min(int,int):

```
int min (const int x, const int y)      // возвращает минимальное значение
{ return x>y ? x : y; }
```

Преобразование из типа в ссылку также тривиально. Его нельзя использовать для различия между перегруженными функциями, поскольку вызовы функции будут выглядеть одинаково. Например, компилятор не сможет отличить следующую функцию от функции int min(int, int):

```
int min (int &x, int &y)            // возвращает минимальное значение
{ return x>y ? x : y; }
```

Но компилятор без труда различает указатели и указываемые типы, например отличает int* от int, а также указатели-константы и не константы от ссылок. В качестве иллюстрации рассмотрим две небольшие и достаточно бессмысленные функции:

```
void printChar (char ch)            // параметр-значение
{ cout << ch; }

void printChar (char* ch)           // параметр-указатель
{ cout << *ch; }
```

Вызовы функции в клиенте выглядят различно: компилятор и программист могут видеть, когда в первом из двух вызовов функции используется обычный символ (не константа) :

```
char c = 'A';  const char cc = 'A';
printChar(c);           // OK, void printChar(char);
printChar(&c);          // OK, void printChar(char*);
printChar(cc);          // константу можно передавать
                        // функции void printChar(char);
printChar(&cc);         // константу нельзя передавать
                        // функции void printChar(char*);
```

Третий вызов также приемлем, так как аргумент типа const может передаваться там, где ожидается отличное от константы значение. Если функция изменяет значение своего параметра, то это изменение не будет распространяться на область действия клиента и не приведет к модификации значения аргумента-константы. Четвертый вызов функции даст синтаксическую ошибку. Если функция изменяет свой параметр (передаваемый по указателю), то это изменение будет распространяться на область действия клиента. Поскольку фактический аргумент объявлен как const, его нельзя использовать в этом вызове функции.

Давайте добавим для комплекта еще одну перегруженную функцию. В этой функции заголовок отражает то, что происходит в ее теле: фактические аргументы данной функцией не изменяются:

```
void printChar (const char* ch)      // указатель, но со значением-константой
{ cout << *ch; }
```

Не все перечисленные выше вызовы функций будут компилироваться и выполняться корректно. Обратите внимание, что если убрать вторую функцию (void printChar(char*);) второй вызов все равно компилируется. Он будет вызывать void printChar(const char*);. Весьма уместно (и безопасно) для передачи отличного от константы значения там, где ожидается значение-константа.

Заметьте также, что все литеральные строки, заключенные в двойные кавычки, имеют тип char*, а не const char*. Вот почему можно установить на них обычные указатели и изменять их через данные указатели:

```
char *p = "day"; p[0] = 'p';                    // теперь здесь "pay"
```

Перегрузку функций можно применять и для функций одного класса. Если число (или типы) параметров должны быть различны, то использование одного имени для разных функций вполне законно. Когда имена функций-членов перегружаются в одном классе, эта семантика должна быть похожей. (Конечно, никаких проверок компилятор в таком случае не выполняет.) Очень часто перегрузка имен используется для конструкторов классов. Оно позволяет в разных контекстах инициализировать объекты в исходном коде клиента (подробнее об этом рассказано в следующей главе).

Хотя приведенные примеры перегрузки имен функций достаточно примитивны, они позволяют продемонстрировать, что данный механизм может затруднить понимание исходного кода клиента. Иногда сложно разобраться, какая именно функция вызывается. Это может представлять трудности и для компилятора C++, и для программиста, сопровождающего приложение. В C++ слишком многое происходит "за кулисами", так что данное средство не стоит применять часто.

Подобно заданным по умолчанию параметрам, перегрузку имен функций можно использовать для развития программы. Когда надо расширить функциональность программы, можно изменить существующие функции согласно новым требованиям. Часто такой подход вынуждает вносить изменения и в интерфейс функции, и в ее тело, а также в вызывающий данную функцию клиентский код. Это сложный, дорогостоящий процесс, чреватый ошибками.

В некоторых случаях перегрузка имен функций дает возможность только добавить новые серверные функции, а не редактировать серверные функции и их вызовы в клиенте. Давайте вернемся назад к простой функции registerEvent(), которая использовалась для иллюстрации применения значений параметров по умолчанию:

```
inline void registerEvent()
{ count++; span = 20; }                    // увеличить счетчик событий
                                           // и задать интервал времени
```

Здесь опять подразумевается, что это сложная и большая система, содержащая около 400 страниц исходного кода, где вызывается данная функция.

```
registerEvent();                  // вызов функции-сервера в клиенте
```

Предположим, понадобилось добавить около 10 страниц исходного кода, где интервал времени устанавливается отдельно для каждого события. 400 страниц исходного кода не требуют внесения изменений, поскольку там интервал времени остается тем же.

Конечно, всегда есть альтернативное решение — написать функцию, например regEvent(), обслуживающую эти 10 страниц кода:

```
inline void regEvent(int duration)        // другая функция-сервер
{ count++; span = duration; }             // увеличить счетчик событий
```

В таком небольшом примере нетрудно написать данную маленькую функцию. В реальной ситуации функции намного сложнее и больше, и всегда легче адаптировать существующие функции к новым условиям. Давайте изменим функцию registerEvent(), добавив дополнительный параметр и изменив соответственно ее тело:

```
inline void registerEvent(int duration)       // изменяем заголовок
{ count++; span = duration; }                 // и тело тоже
```

Как уже говорилось в предыдущем разделе, решение требует:

1. Добавления нового клиентского кода (10 страниц)
2. Изменения заголовка существующей функции-сервера (добавления нового параметра)
3. Изменения тела существующей функции-сервера (использования нового параметра)
4. Модификации имеющегося клиентского кода (все 400 страниц)

Применение назначенных по умолчанию значений устраняет необходимость менять существующий клиентский код, но все равно нужно будет отредактировать тело функции-сервера и ее интерфейс. При перегрузке имен функций менять функцию registerEvent() не потребуется. Достаточно написать другую функцию registerEvent() (она выглядит так же, как последняя функция):

```
inline void registerEvent(int duration)       // новый заголовок функции
{ count++; span = duration; }                 // новое тело функции
```

Таким образом, изменения сводятся к:

1. Добавлению нового клиентского кода (10 страниц)
2. Добавлению новой функции-сервера

Здесь исключаются не только изменения в существующем коде, но и изменения в имеющейся функции-сервере. Замечательно! Не каждая задача поддается такому методу, но, если это так, не упускайте возможности. Это одно из наиболее серьезных усовершенствований в традиционной технологии сопровождения ПО.

Итоги

В данной главе рассмотрены функции C++ как основной инструмент построения программ. Язык C++, развивающий возможности C, является уникальным языком программирования — он требует от программиста включения прототипов функций, применяемых в каждом исходном файле. Данное правило поддерживает раздельную компиляцию и упрощает управление сложными проектами, но создает дополнительные проблемы для разработчиков ПО и сопровождающих его программистов.

Передача параметров — сложная технология C++. Программисту приходится координировать написанный для функции код в четырех местах: в клиенте (вызов функции), в заголовке функции-сервера, в ее прототипе и в теле. Это трудная задача, и иногда возникают ошибки, вызывающие различного рода проблемы.

Передача параметров по значению относительно проста, но не поддерживает изменения фактических аргументов. Передача параметров по указателю поддерживает побочные эффекты в клиентском коде, но сложна и ведет к ошибкам. C++ унаследовал эти два режима передачи параметров из языка C. Чтобы свести к минимуму ошибки, в C++ сделана попытка использовать передачу параметров по указателю как можно реже. Это достигается с помощью еще одного режима — передачи по ссылке. Он кажется хорошим компромиссом, но даже передача по ссылке приводит к некоторому непониманию терминологии и путанице в обозначениях.

Для структур передача по ссылке имеет еще один недостаток: она требует дополнительных времени и памяти для копирования фактических аргументов в стек (для параметра функции выделяется память в стеке). Передача по ссылке устраняет копирование данных и не усложняет процесс, как это происходит в случае передачи по указателю. Но при передаче параметра по ссылке трудно сообщить намерения разработчика программисту, сопровождающему приложение, указать, какие параметры модифицируются функцией, а какие нет. Применение спецификатора `const` решает данную проблему. Это чрезвычайно полезная техника.

Для массивов доступен только один режим, а для входных и выходных параметров синтаксис совпадает. При этом затрудняется понимание потока данных в программе сопровождающим программистом: не ясно, какие именно параметры модифицируются функцией, а какие сохраняют свои значения.

Использование спецификаторов `const` позволяет разработчику передать программисту информацию о том, что массивы не изменяются в результате вызова функции. Предположение, что массивы без модификатора `const` обязательно изменяются функцией, не всегда имеет под собой основание, и, чтобы эта техника действительно была полезна программисту, сопровождающему приложение, разработчик должен действовать осторожно.

Мы рассмотрели также преобразование и приведение типов аргументов. Когда типы аргументов и параметров несовместимы, преобразование не допускается. Типы не совместимы, если они принадлежат к разным категориям (скалярное значение, структура, указатель, массив). Преобразование между категориями не допускается. В этом плане в C++ соблюдается строгий контроль типов. Между тем, C++ допускает неявное преобразование между скалярными числовыми типами, не задавая лишних вопросов. Кроме того, C+ поддерживает явное преобразование (приведение) между указателями или массивами разных типов. Эти преобразования предоставляют программисту более гибкие возможности, но способствуют ошибкам и могут затруднять понимание и сопровождение программы.

Мы обсудили также встраиваемые функции, позволяющие избавиться от непроизводительных издержек вызова функции. При корректном использовании эти функции могут повышать производительность программы, а при некорректном способны увеличить размер объектного кода и даже снизить производительность из-за избыточного свопинга.

Кроме того, в данной главе рассказывалось о присваивании параметрам значений по умолчанию и о перегрузке имен функций. Это превосходные средства языка, уменьшающие ограничения пространств имен в программных проектах C++. Они открывают также новые перспективы в сопровождении программ, позволяя обойтись без изменения существующего клиентского кода, когда вызываемые из этого кода функции требуют изменений. Между тем данные средства должны использоваться по возможности реже. Они сложны, и слишком многое происходит в программе C++ "за кулисами". Непродуманное использование данных возможностей может запутать не только компилятор, но и сопровождающего ПО программиста.

Методы программирования функций составляют основу C++. Не освоив на хорошем уровне функции C++, невозможно создавать высококачественные объектно-ориентированные программы. На самом деле, без этого нельзя создавать любые качественные программы — объектно-ориентированные или нет. Следующая глава начинается с изучения объектно-ориентированного программирования — самого мощного способа создания высококачественных программ.

Глава 8

Объектно-ориентированное программирование с использованием функций

Темы данной главы

- ✔ Сцепление
- ✔ Связность
- ✔ Инкапсуляция данных
- ✔ Сокрытие информации
- ✔ Большой пример инкапсуляции
- ✔ Недостатки инкапсуляции в функциях
- ✔ Итоги

С этой главы начинается обсуждение принципов и методов объектно-ориентированного программирования. Некоторые из них можно отнести к обычным навыкам программирования, другие сформулированы и адаптированы для использования с C++. Такие принципы и методы редко обсуждаются в других книгах по C++, поэтому даже программистам, уже имеющим опыт работы на C++, не стоит пропускать данную главу.

В предыдущих главах основное внимание уделялось правилам языка C++, определяющим, что синтаксически допустимо и что недопустимо в языке C++. Подобно естественным языкам, недопустимые конструкции должны исключаться не только из соображений их неоднозначности или плохого стиля, а просто потому, что компилятор не сможет преобразовать их в объектный код. Что касается допустимых конструкций, то они позволяют выразить одно и то же разными способами. В предшествующих главах сравнивались разные способы применения различных конструкций — часто с точки зрения корректности программы, ее производительности, и, конечно, стиля. Между тем основным вопросом было удобство сопровождения программы — нужно добиться того, чтобы сопровождающий ее программист не тратил лишние усилия на попытки понять, что имел в виду разработчик, когда писал исходный код.

В этой главе (и в следующих главах) понятность исходного кода также будет важным вопросом. Но фокус дискуссии сместится с написания управляющих конструкций в исходном коде на более высокий уровень программирования: разбиение программы на взаимодействующие части (функции и классы).

Не будем углубляться в системный анализ и разбираться в том, какие функции должны присутствовать в приложении для достижения поставленных целей. Это чрезмерно расширило бы тему данной книги. Будем предполагать, что необходимые функции для достижения целей программы уже имеются, и сконцентрируемся на способах использования дополнительных функций, улучшающих удобство сопровождения и повторного использования программы.

Работу между функциями-клиентами, взаимодействующими для достижения целей программы, всегда можно разделить несколькими способами. Есть также несколько способов проектирования серверных функций, обрабатывающих данные и выполняющих операции по запросам функций-клиентов. Если предположить, что все версии эквивалентны с точки зрения корректности программы, то как выбрать лучшую?

Ранее большинство программистов в качестве критерия руководствовались производительностью программы. Прогресс в области аппаратного обеспечения сделал этот критерий неподходящим для большинства приложений, особенно для интерактивных. Для тех приложений, где производительность все еще важна, выбираются влияющие на быстродействие алгоритмы и структуры данных, а не способ распределения работы между клиентскими и серверными функциями.

Еще один важный критерий — простота написания программного кода. Этот критерий до сих пор подходит и для небольших программ, разрабатываемых несколькими программистами и используемых непродолжительное время (после чего они заменяются новыми), и для крупных систем, эксплуатируемых очень долго, в создании которых участвуют большие коллективы разработчиков. В то же время экономика разработки ПО предполагает в этих случаях другой подход. Лучшая версия программы — та, у которой отдельные части можно использовать повторно и делать это легче (что предполагает экономию при разработке приложения и создания следующих версий), или та версия, которая проще в сопровождении (что предполагает экономию при развитии и совершенствовании программы).

Удобство сопровождения и повторного использования — *это* две наиболее важные характеристики качества ПО. Однако эти характеристики слишком общие. Вовсе не очевидно, какую версию кода легче и дешевле сопровождать, а какую проще использовать повторно.

Возможность повторного использования тесно связана с независимостью отдельных частей программы. Среди нескольких версий кода C++ версию, в которой разобраться проще и быстрее (предпочтительнее, не обращаясь к другим сегментам программы), как правило легче изменять без нежелательного влияния на другие фрагменты кода.

Таким образом, необходимость ссылаться на другие сегменты программы свидетельствует о плохом качестве кода, а возможность изолированного анализа исходного кода без ссылок на другие сегменты программы говорит о хорошем его качестве. Поэтому будем говорить, что одна версия кода лучше, чем другая, если она более понятна, т. е. чтобы разобраться в ней, требуется меньше усилий и обращений к другим частям программы.

Все это хорошо, но для программиста-практика недостаточно специфично и точно. Концепции "понятности" и "независимости" должны поддерживаться более специфическими техническими критериями, которые легче распознавать и использовать. В данной главе предлагается несколько технических критериев. Два из них — сцепление и связность — относительно стары, а два других — инкапсуляция и сокрытие информации — довольно новы, и отрасль не накопила достаточно опыта их использования. Кроме инкапсуляции и сокрытия информации, будем использовать несколько разновидностей критериев, связанных с понятностью и независимостью кода:

- Перенос обязанностей с функции-клиента на функцию-сервер
- Ограниченность знания, используемого клиентом и сервером

- Разделение задач клиентской и серверной функции
- Не разделение тех частей, которые должны быть вместе
- Передача знания разработчика сопровождающему приложение программисту в самой программе, а не в комментариях

Никакого всеохватывающего термина для этих принципов подобрать не удалось (принцип максимальной независимости?; принцип Штерна?; разделения знания по принципу необходимости?; принцип самодокументируемого кода?). Как будет понятно дальше, данные принципы в чем-то перекрещиваются друг с другом и с критериями сцепления, связности, сокрытия информации и инкапсуляции. Практикующие программисты должны быть знакомы со всеми перечисленными принципами. Их основное достоинство состоит в том, что все они применимы в работе и показывают, в каком направлении нужно двигаться, чтобы улучшить архитектуру программы и ее качество, как нужно усовершенствовать методы программирования.

В основе данных критериев лежит идея, что функции программы взаимодействуют друг с другом, выполняя части общей работы. Как бы ни распределялись между ними обязанности, функции должны использовать какие-то общие знания, иметь общие цели, работать над частью одной задачи. Все это производят разные функции, но они — части одной программы. Чтобы сделать данные функции понятными, чтобы их можно было повторно использовать, нужно так распределить между ними обязанности, спроектировать систему таким образом, чтобы зависимости между функциями были минимальными.

Как это часто бывает, написание программы более высокого качества требует дополнительных усилий, а программа содержит больше строк, чем менее качественная программа. Некоторые программисты (и менеджеры) будут, наверное, разочарованы таким увеличением объема работы. Но можно привести интересную аналогию с правилами дорожного движения.

Когда я стою на красном сигнале светофора, то иногда думаю, что без ограничивающих правил дорожного движения добрался бы до места быстрее. Возможно, это и так, но не для всех мест назначения и не для всех водителей. Езда без правил приведет к авариям и пробкам на дорогах. Водители, избежавшие аварий и пробок, могут действительно добраться до места быстрее. Но многие другие попадут в пункт назначения значительно позже ожидаемого времени. Правила движения отнимают у нас время, чтобы, в конечном счете, сэкономить его.

Аналогично игнорирование правил удобства сопровождения и повторного использования программы позволит написать ее быстрее, но так будет не для всех приложений и не для всех программистов. Время, сэкономленное на написании программы, будет существенно меньше времени, которое придется потратить, чтобы разобраться в ней и понять, каких целей стремились добиться разработчики (и где они ошиблись).

Вот почему в индустрии ПО столь большое внимание уделяется написанию комментариев. Комментарии в программе — это своего рода инвестиции, в конечном счете окупающие себя (когда они ясные, полные и не устаревшие). Между тем часто строки комментариев неполны, непонятны и не отражают изменений, внесенных после написания программы. Затраты на написание понятного программного кода предпочтительнее затрат на комментарии.

При написании небольшой программы правила создания качественного, понятного кода не очень важны, но если разрабатывается большое приложение, то затраты на разработку качественного кода имеют решающее значение и в результате дадут отдачу.

Сцепление

Сцеплением называют связанность шагов, реализованных в одном сегменте кода, например в функции.

Если функция обладает сильным сцеплением (high cohesion), то она выполняет одну задачу с одним вычислительным объектом или структурой данных. При слабом сцеплении функция выполняет несколько задач с одним объектом или даже несколько задач с несколькими объектами. Функция со слабым сцеплением включает в себя вычисления, не имеющие отношения друг к другу и выполняемые с независимыми объектами. Это означает, что разработчик совместил в одной функции шаги, которые не должны выполняться вместе.

Функциям с сильным сцеплением проще давать имена. Обычно используется комбинация "глагол + существительное". Глагол обозначает выполняемое данной функцией действие, а существительное — объект (субъект) действия. Например, `insertItem()`, `findAccount()` и т. д. (если, конечно, имя функции соответствует ее содержанию, что бывает не всегда).

Для функций со слабым сцеплением пришлось бы использовать несколько глаголов или существительных, например `findOrInsertItem()`.

Вот еще один пример, хотя и несколько неуклюжий (все хорошие примеры функций со слабым сцеплением неуклюжие, так как описывают плохо спроектированные функции):

```
void initializeGlobalObjects ()
{ numaccts = 0;                              // один вычислительный объект
  fstream inf("trans.dat",ios::in);          // файл транзакций
  numtrans = 0;                              // еще один вычислительный объект
  if (inf==NULL) exit(1); }                  // снова файл транзакций
```

В данном примере переменную `numaccts` следует инициализировать там, где обрабатывается `accounts` (где выполняются операции со счетами). Аналогично `numtrans` нужно инициализировать при обработке транзакций, а не при инициализации счетов. В этой функции разделено и включено в функцию со слабым сцеплением то, что должно быть вместе (т. е. группироваться с другими шагами обработки).

Преодолеть проблему можно, спроектировав функцию заново. Как уже упоминалось в главе 1, перепроектирование означает изменение списка отдельных частей (функций) и их обязанностей. В случае слабого сцепления перепроектирование обычно предполагает разбиение функций со слабым сцеплением на несколько функций с сильным сцеплением. В результате число функций может значительно увеличиться. Кроме потенциального негативного влияния на производительность, это затруднит сопровождение программы — специалистам по сопровождению придется очень много запоминать (имена функций и их интерфейсы). Для маленьких функций, таких, как `initializeGlobalObjects()`, разбиение не имеет смысла. Вероятно, такие функции следует исключить.

Сцепление нельзя считать очень строгим критерием. Решение о перепроектировании и разделении функций — не абсолютное требование. В случае сомнения следует руководствоваться другими критериями. Сцепление важно для оценки проекта. Им нужно руководствоваться при выборе альтернативных вариантов — распределении работ между функциями.

Связность

Связность (coupling) — намного более сильный и полезный критерий, чем сцепление. Он описывает интерфейс или поток данных между вызываемой функцией (функцией-сервером) и вызывающей функцией (функцией-клиентом).

Связность может быть неявной (когда функции взаимодействуют через глобальные переменные) или явной (клиент и сервер взаимодействуют через параметры). Неявная связность дает более сильную зависимость между функцией-клиентом и функцией-сервером. Неявная связность предпочтительнее: при коммуникациях через параметры функции легче понять, повторно использовать и модифицировать.

Степень связности определяется числом значений, передаваемых от клиентской функции серверной и обратно. Большое число значений означает сильную связность (высокую степень зависимости между функциями), а малое число значений означает слабую связность (низкую степень зависимости между функциями (клиентом и сервером).

Неявная связность

Клиентская функция передает серверной функции входные данные и использует вычисленный серверной функцией результат (выходные значения сервера). Связность неявна, когда коммуникации с функцией осуществляются через глобальные переменные, не перечисленные в интерфейсе функции.

Рассмотрим, например, интерактивную программу, которая просит пользователя ввести год и выводит сообщение о том, високосный ли он.

```
int year, remainder; bool leap;            // данные программы
cout << "Введите год: ";                   // приглашение пользователю
cin >> year;                               // получение данных от пользователя
remainder = year % 4;
if (remainder != 0)                        // не делится на 4
    leap = false;
else
  { if (year%100 == 0 && year%400 !=0)
       leap = false;                       // делится на 100, но не на 400
    else
       leap = true; }                   // в противном случае это високосный год
if (leap)
  cout << year << " високосный год\n";      // вывод результатов
else
  cout << year << " не високосный год\n";
}
```

Эта программа аналогична той, которая уже обсуждалась в главе 4 (листинги 4.8 и 4.9). Она небольшая и не нуждается в разбиении на модули. Но программа, где модульность действительно дает преимущества, должна быть достаточно большого размера. Детальное изучение таких программ и сравнение альтернативных вариантов само по себе могло бы стать нелегкой задачей и отвлечь нас от принципов модульности, на которых сейчас стоит сконцентрироваться. Ведь именно сами эти принципы, а не детали различных примеров следует применять в реальной ситуации.

Давайте представим, что это большая и сложная программа, и займемся ее перепроектированием, разбив на взаимодействующие функции.

Итак, мы имеем монолитную программу, которую желательно разделить на управляемые компоненты. Для простоты разделим ее только на две функции: функцию `main()`, отвечающую за интерфейс с пользователем и общие вычисления, и функцию `isLeap()`, использующую значения `year` и `remainder` для вычисления значения `leap`, на основе которого `main()` выводит результат.

```
void isLeap()
  { if (remainder != 0)                    // не делится на 4
       leap = false;
```

```
else if (year%100 == 0 && year%400! = 0)
   leap = false;                    // делится на 100, но не на 400
else
   leap = true; }                   // в противном случае это високосный год
```

Здесь есть одна техническая проблема, которая относится к обсуждавшейся в главе 6 концепции области действия. Значения year и remainder, используемые функцией isLeap(), устанавливаются в функции main(). Вычисляемое функцией isLeap() значение leap применяется в main(). Однако, если определяем эти переменные в main(), они будут видимы только там. Правила области действия C++ предотвращают "видимость" этих значений в любых других функциях, и isLeap() не сможет манипулировать данными переменными. Если определить эти переменные в isLeap(), они будут видимы только в функции isLeap(). Правила области действия C++ сделают их невидимыми в функции main(). Чтобы переменные были доступны и в main(), и в isLeap(), для обеих функций эти переменные нужно определить как глобальные.

```
Введите год: 1999
1999 не високосный год
```

Рис. 8.1. *Результат программы из листинга 8.1*

Это решение демонстрируется в листинге 8.1. Пример выполнения программы показан на рис. 8.1.

Листинг 8.1. Пример неявного связывания через глобальные переменные

```
#include <iostream>
using namespace std;

int year, remainder;                    // глобальные переменные (вход)
bool leap;                              // глобальная переменная (выход)

void isLeap()
  { if (remainder != 0)                 // не делится на 4
      leap = false;
    else if (year%100==0 && year%400!=0)
      leap = false;                     // делится на 100, но не на 400: не високосный год
    else
      leap = true; }                    // в противном случае это високосный год

int main()
{ cout << "Введите год: ";
  cin >> year;                          // получение данных от пользователя
  remainder = year % 4;
  isLeap();
  if (leap)
    cout << year << " високосный год\n";    // вывод результатов
  else
    cout << year << " не високосный год\n";
  return 0;
  }
```

В данной программе функция main() вызывает функцию isLeap(). Функция main() является клиентом и выполняет свою задачу, вызывая другие функции. Функция isLeap() — это сервер. Она делает некую работу для вызывающего ее клиента. Соотношение между функциями показано на рис. 8.2. Эта структурная диаграмма демонстрирует потоки данных между функциями. Переменные year и remainder устанавливаются в функции main() и используются в функции isLeap() как входные значения для вычисления результата. Подсчитанное функцией isLeap() значение переменной leap — это ее выходное значение. Оно используется функцией main() после вызова isLeap().

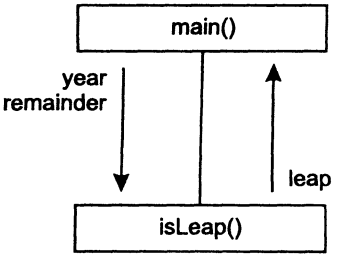


Рис. 8.2. *Структурная диаграмма для программы из листинга 8.1*

Заметим, что перед вызовом функции isLeap() в функции main() входные переменные year и remainder должны иметь допустимые значения. Функция-клиент должна убедиться, что эти значения правильно инициализированы. Функция-сервер isLeap() не проверяет допустимость значений. Она предполагает, что функция main() исполняет свои обязательства.

Аналогично выходные переменные (в данном случае leap) не обязаны содержать допустимое значение перед вызовом функции-сервера isLeap(). Эта функция сама должна установить выходное значение, а клиент — позднее, после вызова (но не перед ним), его использовать.

Очень важно представлять поток данных между функциями. Если известно, что переменные year и remainder являются входными переменными функции isLeap(), то можно ожидать, что функция-сервер использует эти значения, но не изменяет их. Было бы крайне странно предполагать, что функция isLeap() делает что-то вроде следующего:

```
void isLeap()
  { remainder = 4; year = 2000; ...              // нонсенс!
```

Кроме того, если известно, что переменная leap — выходная переменная функции isLeap(), то не стоит ожидать, что клиент main() инициализирует эту переменную перед вызовом isLeap() или будет изменять ее значение сразу после вызова, предварительно не использовав его для тех или иных целей.

```
int main()
{ cout << "Введите год: ";
  cin >> year;                          // получение данных от пользователя
  remainder = year % 4;
  leap = false;
  isLeap();
  leap = true;                          // вводит в заблуждение (и некорректно),
                                        // если выполняется сразу после вызова
  . . .
```

Что будет думать сопровождающий приложение программист, прочитав приведенную выше функцию? После определения цели присваивания remainder (эта переменная используется в isLeap() для вычисления значения переменной leap), программисту придется снова исследовать функцию isLeap() и попытаться понять, для чего выполняется присваивание leap. Для маленькой функции достаточно нескольких секунд, чтобы сделать вывод: значение, присвоенное в клиенте main() переменной leap, не используется функцией-сервером isLeap() и даже самим клиентом main(). Но это лишь для маленькой функции. Для крупной программы потребуется гораздо больше времени. Сопровождающий ее программист может запутаться и сделать неверные выводы.

Действительно, некоторые программисты настолько не любят неинициализированных переменных, что инициализируют их, даже когда в том нет необходимости. По их мнению, это помогает, когда функция-сервер по тем или иным причинам не присваивает значение. Однако isLeap() не относится к таким функциям! Как и большинство других функций. Если программисты понимают поток данных между функциями, то не возникает ситуация, когда функция не присваивает значения выходной переменной.

Как видно, такая невинная на первый взгляд "защитная" мера программирования дает в результате код, для понимания которого требуется больше времени. С точки зрения критерия качества (удобства чтения программы и независимости отдельных ее частей) эта техника неизбежно дает худший код, т. е. является прямым вкладом в кризис ПО, который мы хотим преодолеть. Избегайте такой практики. Вместо инициализации всего подряд нужно сообщить сопровождающему

приложение программисту, какие значения будут использоваться сервером в качестве ввода (инициализируя их в клиенте), а какие являются выходными переменными сервера (не инициализируя их).

Надеюсь, вы следите за дискуссией и понимаете важность передачи сопровождающему приложение программисту знания разработчика о потоке данных между функциями. Давайте вернемся к обсуждению связности.

Связность определяет, сколько усилий и времени потребуется для понимания потока данных между функциями. Часто для этого необходимо исследовать обработку данных клиентом и функцией-сервером. Например, в листинге 8.1 функция `main()` присваивает значения переменным `year` и `remainder`, а `isLeap()` использует эти значения, а также что `main()` не инициализирует `leap`, `isLeap()` присваивает значение `leap`, а `main()` использует это значение после вызова `isLeap()`. Все так.

Однако, чтобы выявить эти простые зависимости, надо изучить функцию-клиент и функцию-сервер во всей полноте. В таком тривиальном примере это сделать легко, но в более реалистичной и сложной функции значительного размера потребуется гораздо больше времени. Можно ли усовершенствовать данную трудоемкую и подверженную ошибкам технику? Конечно. С помощью явной связности.

Явная связность

Явная связность осуществляется через параметры функции: все переменные (вход и выход), используемые функцией-сервером, включаются в параметры этой функции, и глобальные переменные в потоке данных между клиентом и сервером не используются. Листинг 8.2 показывает тот же пример, что и в листинге 8.1, но неявный поток данных через глобальные переменные заменен на явные параметры. Эта программа выполняется аналогично программе из листинга 8.1.

Листинг 8.2. Пример явного связывания через параметры

```
#include <iostream>
using namespace std;

void isLeap(int year, int remainder, bool &leap)          // параметры
// ввод: year, remainder; вывод: leap
  { if (remainder != 0)
      leap = false;
    else if (year%100==0 && year%400!=0)
      leap = false;
    else
      leap = true; }

int main()
{ int year, remainder;                                    // локальные переменные (ввод)
  bool leap;                                              // локальная переменная (выход)
  cout << "Введите год: ";
  cin >> year;                                            // получение данных от пользователя
  remainder = year % 4;
  isLeap() (year, reminder, leap);
  if (leap)
    cout << year << " високосный год\n";
  else
    cout << year << " не високосный год\n";
  return 0;
  }
```

В листинге 8.2 функция-сервер isLeap() имеет три параметра. Это не глобальные переменные. Переменные year, remainder и leap определяются в функции-клиенте main() как локальные. Почему это возможно? Потому что они не должны быть известны в области действия функции isLeap(), как в листинге 8.1. Вместо этого функция isLeap() обращается к данным переменным как к фактическим аргументам — они передаются в вызове функции isLeap().

Можно сделать следующий общий вывод: когда две функции взаимодействуют друг с другом через данные, компоненты потока данных должны либо описываться как глобальные переменные, либо определяться в области действия клиентской функции и передаваться серверной функции как параметры.

Как и в предыдущем примере, переменные year и remainder являются для функции isLeap() входными, а leap — выходная переменная. Откуда это известно? Достаточно взглянуть на заголовок (или прототип, если он используется) функции isLeap(), а не на тело функции:

```
void isLeap(int year, int remainder, bool &leap)        // параметры
{ . . . }
```

Можно ли сказать, не изучая тела функции, какова роль каждого параметра? Конечно. Параметры year и remainder передаются по значению. Следовательно, они не могут быть выходными параметрами, и функция isLeap() не может устанавливать их значение.

```
void isLeap(int year, int remainder, bool &leap)        // параметры
{ remainder=4; year=2000; . . .          // бесполезно для значения параметров
```

Следовательно, можно сделать вывод, что это входные параметры. Значения фактических параметров должны устанавливаться в коде клиента перед вызовом функции, и эти значения будут использоваться функцией-сервером в вычислениях.

Аналогично параметр leap передается по ссылке. Это означает, что данный параметр выходной. На самом деле он может быть параметром ввода-вывода, т. е. функция-клиент может сначала устанавливать его значение, а функция-сервер — обновлять его. Но основная идея в том, что функция isLeap() изменяет значение параметра leap.

Как прийти к таким выводам? Для этого достаточно взглянуть на заголовок функции. Структурная диаграмма программы из листинга 8.2 показана на рис. 8.2. Она аналогична программе из листинга 8.1, но явный поток данных через глобальные переменные заменен на явный поток данных через параметры. Зависит ли потраченное время от размера и сложности функции-клиента? Нет. А от сложности серверной функции? Нет. Переход от неявной связности к явной дает значительное уменьшение сложности исходного кода как с точки зрения разработчика, так и с точки зрения сопровождающего приложение программиста.

Данный пример показывает, почему следует избегать глобальных переменных. Вот уже почти 30 лет прошло с тех пор, как в индустрии ПО впервые начались дискуссии по использованию глобальных переменных, но многие программисты до сих пор не уяснили суть проблемы. Они считают, что любая функция в файле (или даже в программе) может случайно (или умышленно) изменить значение глобальной переменной и в результате очень трудно будет найти источник ошибки. Некоторые добавляют: существо проблемы в том, что вовсе не очевидно, какие именно функции обращаются к данной глобальной переменной. Это означает, что проблема может возникнуть в любом месте программы.

Возможно, все сказанное и верно (правда, есть некоторые сомнения насчет важности несанкционированного доступа к переменным), но основной ущерб от применения глобальных переменных — это неявное связывание. Использование глобальных переменных вынуждает разработчика и сопровождающего приложение программиста изучать большие сегменты кода, чтобы понять поток данных в программе. Применение явного связывания через параметры позволяет получить

представление о потоке данных, исследовав лишь заголовки функции-сервера (или прототипы). Как говорится, почувствуйте разницу.



Советуем Избегайте неявного связывания через глобальные переменные. Используйте явное связывание через параметры. В результате разработчик (и программист, работающий с функцией-клиентом, вызывающим функцию-сервер) может понять интерфейс функции, исследовав лишь ее заголовок, а не все тело функции и вызывающую ее программу.

Однако такое снижение сложности не достигается автоматически, лишь благодаря использованию явного связывания через параметры вместо глобальных переменных. Следует корректно выбирать режимы передачи параметров. Рассмотрим, например, следующую версию функции-сервера `isLeap()`:

```
void isLeap(int &year, int &remainder, bool &leap)          // параметры
  { if (remainder != 0)
       leap = false;
    else if (year%100==0 && year%400!=0)
       leap = false;
    else
       leap = true; }
```

Корректна ли она синтаксически? Да. А семантически? Да. Если использовать эту функцию вместо той, которая применяется в листинге 8.2, то результаты будут такие же, причем для любого набора входных данных.

Хороша ли эта функция с точки зрения качества ПО? Нет. Все ее параметры передаются по ссылке, что вводит в заблуждение сопровождающего приложение программиста: он думает, что их значения устанавливаются в функции и используются клиентом. Чтобы выяснить истину, программисту нужно изучить серверную функцию целиком. Это лучше, чем изучать и клиентский, и серверный код, как в случае использования только глобальных переменных, но гораздо удобнее исследование одного лишь серверного интерфейса, как в листинге 8.2.

Передавая в этой версии функции все параметры по ссылке, разработчик функции не может на этапе проектирования функции сообщить о том, что именно он задумал. Он знает, что параметр `leap` — единственный выходной параметр, но не может обозначить это в самом программном коде.

Сопровождающий приложение программист должен поверить, что передача по ссылке предусматривает изменение параметра функцией-сервером (если отсутствует модификатор `const`), а передача по значению говорит о том, что параметр не изменяется. В противном случае ему придется изучать клиентскую и серверную функции во всех деталях, а не только просматривать список параметров сервера. В такой ситуации все преимущества явной связности сводятся на нет.

Таким образом, правила, сформулированные в главе 7, очень важны. Постоянное следование им позволяет описать интерфейс функции для сопровождающего программиста, устраняет необходимость изучать несколько функций сразу, уменьшает объем кода, подлежащий исследованию. Режимы передачи параметров следует выбирать корректно. Наличие модификатора `const` свидетельствует, что это входной параметр. Отсутствие `const` говорит о том, что параметр изменяется функцией. Не пренебрегайте этим мощным методом повышения качества ПО.

Если применение параметров настолько лучше, чем использование глобальных переменных, то почему же программисты до сих работают с глобальными переменными? На то есть три причины.

Первая — производительность программы. Функции, использующие параметры, тратят время на распределение и освобождение памяти для этих параметров и копирование их значений (или значений адресов). При применении глобальных

переменных функции работают быстрее. Если вы используете глобальные параметры для данной цели, заранее продумайте два момента. Во-первых, программа в самом деле должна сталкиваться с проблемой производительности. Во-вторых, следует убедиться в том, что применение глобальных переменных для этой цели действительно устранит проблему. Подчеркнем: реально *знать*, а не *думать*, что глобальные переменные ускорят работу программы.

Применение глобальных переменных с редко вызываемыми функциями не уменьшит времени выполнения программы. Их использование не повлияет и на функции с внешним вводом и выводом. В коротких и простых функциях глобальные переменные также не увеличат скорость выполнения программы, поскольку эти функции мало влияют на время выполнения программы в целом. Это не говорит о том, что не стоит использовать глобальные переменные вовсе, но нужно действительно знать, когда они помогут увеличить быстродействие программы.

Вторая причина применения глобальных переменных — производительность разработчика. Намного легче и быстрее написать серверную функцию, где используются глобальные переменные, а не параметры. При применении параметров (как в листинге 8.2) легко может оказаться, что введены дополнительные параметры, в которых на самом деле нет необходимости, или наоборот, число параметров следует увеличить, что вынуждает возвращаться к функции и переписывать ее. Написание функции с параметрами связано с дополнительным временем на предварительное планирование.

В листинге 8.1 необходимые для функции глобальные переменные определяются и используются без дополнительного планирования. Когда-то это считалось важным преимуществом. Полагали, что ускорение разработки программного кода имеет критическое значение. Сегодня специалисты уже не считают, что облегчение *написания* программного кода экономит время и деньги. Эта экономия достигается за счет упрощения *чтения* программы, и современные языки, в том числе и C++, ориентированы на то, чтобы побуждать разработчика тратить больше времени на создание легко читаемого исходного кода.

Третья причина применения глобальных переменных для коммуникаций между функциями — недостаточные знания программистов. Они не особенно задумываются о сложности использования глобальных переменных в серверных функциях и просто применяют их. Тем самым увеличивается необходимость взаимодействия с другими разработчиками, однако программисты не утруждают себя мыслями о том, что такое взаимодействие влияет на качество программного кода.

Поясняемые здесь вопросы редко обсуждаются в книгах по программированию. Некоторые из них освещаются в книгах по программной инженерии, но в них обычно представлены лишь общие принципы, а не конкретные приемы программирования на том или ином языке. Надеюсь, что данное обсуждение, наряду со сказанным в главе 7, убедит вас в том, что:

- Использовать параметры функций лучше, чем глобальные переменные
- Нужно передавать простые входные параметры по значению, а выходные — по ссылке
- Следует передавать параметры-структуры и классы по ссылке, применяя для входных параметров модификатор `const`
- Необходимо передавать выходные параметры, используя модификатор `const` (и выходные массивы без `const`)



Осторожно! Передавайте параметры с соблюдением рекомендаций, приведенных в данной книге. Отклонение от этих рекомендаций упрощает написание программного кода, но скрывает от сопровождающего приложение программиста намерения разработчика, т. е., какие параметры функции являются входными, а какие — выходными.

Как уменьшить степень связности

Степень связности функций определяется числом значений в потоке данных между клиентом и сервером. Чем больше этих значений, тем больше клиент и сервер зависят друг от друга и тем труднее изучить одну функцию, не изучая другую.

Как уменьшить поток данных между функциями? Непростая задача. Единственный способ свести к минимуму зависимость между функциями состоит в их перепроектировании, т. е. разделении обязанностей между ними. Все другие подходы тщетны.

Например, некоторые программисты полагают, что число параметров можно уменьшить, скомбинировав их в структуру. В чем-то они правы. Количество параметров при этом действительно уменьшается, однако это вовсе не обязательно ведет к уменьшению связности. В листинге 8.3 показана версия функции `isLeap()`, у которой три параметра скомбинированы.

Листинг 8.3. Пример объединения параметров в структуру

```
#include <iostream>
using namespace std;
struct YearData
{ int year, remainder;
  bool leap; } ;

void isLeap(YearData &data)                    // только один параметр
   { if ( data remainder != 0)
        data.leap = false;
     else if (datd.year%100==0 && data year%400!=0)
        data.leap = false;
     else
        data.leap = true; }

int main()
{ YearData data;                               // локальная переменная
  cout << "Введите год: ";
  cin >> data.year;                            // присваивание выходных полей
  data.remainder = data.year % 4;
  isLeap(data);
  if (data.leap)
     cout << data.year << " високосный год\n";
  else
     cout << data.year << " не високосный год\n";
  return 0;
  }
```

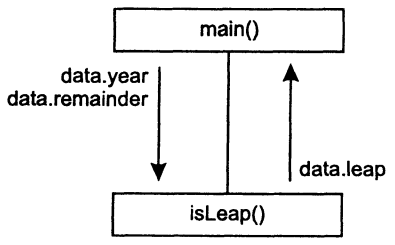


Рис. 8.3. *Структурная диаграмма и поток данных для программы из листинга 8.3*

Число параметров здесь в самом деле меньше, чем в листинге 8.2, но уменьшился ли поток данных между функциями? Поток данных для этой версии программы показан на рис. 8.3. Как видно, входных значений `getVolume(c1)` все равно два — `data.year` и `data.remainder`, а выходное значение одно — `data.leap`.

Можно даже сказать, что такую версию программы сложнее писать, она, определенно, труднее в понимании и ее сложнее повторно использовать, так как данную версию `isLeap()` нельзя применять без типа `YearData`. В любом случае основной вывод в том, что такая версия программы не уменьшает связности между

функциями. Это естественно. При подготовке данной версии никакого перепроектирования не выполнялось — в ней обязанности между функциями main() и isLeap() распределены точно так же, как в версии из листинга 8.2. Следовательно, поток данных между ними остался тем же.

Некоторые программисты пытаются уменьшить связность, избегая выходных параметров, и считают, что для этого нужно применять возвращаемое функцией значение. И они в чем-то правы. В листинге 8.4 показана еще одна версия данной программы. В ней функция isLeap() возвращает значение, а не присваивает его выходному параметру leap.

Листинг 8.4. Пример использования возвращаемого значения вместо выходного параметра

```
#include <iostream>
using namespace std;

bool isLeap(int year, int remainder)           // меньше параметров
  { if (remainder != 0)
       return = false;
    else if (year%100==0 && year%400!=0)
       return false;
    else
       return true; }

int main()
{ int year, remainder;                         // локальная переменная (вход)
  bool leap;                                   // локальная переменная (выход)
  cout << "Введите год: ";
  cin >> year;                                 // присваивание входных переменных
  remainder = year % 4;
  leap = isLeap(year,reminder);
  if (leap)
     cout << year << " високосный год\n";
  else
     cout << year << " не високосный год\n";
  return 0;
  }
```

Здесь число параметров в потоке данных меньше, чем в листинге 8.2. Функция isLeap() в данном случае проще в написании, и нет необходимости бороться с параметром-ссылкой. Например, можно вовсе устранить переменную leap, непосредственно используя в операторе if функции main() возвращаемое функцией isLeap() значение, а не устанавливая сначала значение локальной переменной:

```
int main()
{ int year, remainder;                         // нет переменной leap
  cout << "Введите год: ";
  cin >> year;                                 // присваивание входных переменных
  remainder = year % 4;
  if (isLeap(year,remainder)==true)            // используется выходное значение
     cout << year << " високосный год\n";
  else
     cout << year << " не високосный год\n";
  return 0; }
```

Уменьшился ли в программе поток данных между функциями main() и isLeap()? Не совсем. Рис. 8.4 показывает поток данных для данной версии программы. Как видно, здесь все равно два входных значения — year и remainder, а выходное значение представлено значением, возвращаемым функцией.

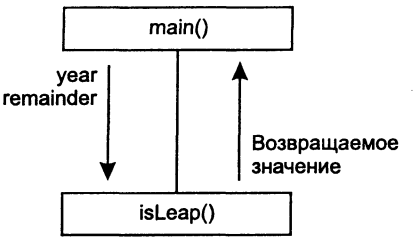


Рис. 8.4. *Структурная диаграмма и поток данных для программы из листинга 8.4*

Связность здесь также не уменьшилась, поскольку перепроектирование не выполнялось. Эта программа распределяет обязанности между функциями main() и isLeap() так же, как в версии из листинга 8.2.

Чтобы уменьшить связность, следует проанализировать распределение вычислений между функциями и применить принципы, перечисленные в начале главы. Это можно сделать, например, идентифицировав компоненты потока данных, которые следует разделить или объединить. Разделение вычислений вместо объединения их в одной функции обычно требует организовать дополнительное взаимодействие между частями программы. Если такие вычисления реализуются в разных функциях, хотя могли бы совмещаться в одной, получается избыточный поток данных. Объединение в одной функции опрометчиво разделенных операций исключает лишние коммуникации между функциями.

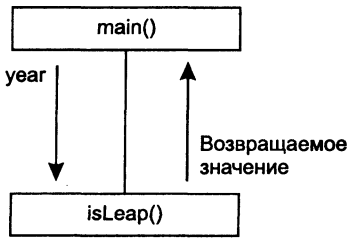


Рис. 8.5. *Структурная диаграмма и поток данных для программы из листинга 8.5*

Иногда такое ненужное разделение происходит в результате непонимания смысла параметра после исследования кода серверной функции без изучения клиента. Например, в листинге 8.4 смысл параметра remainder невозможно уяснить только из функции isLeap(). Программисту нужно исследовать клиентскую функцию main() и сделать вывод, что данная переменная представляет остаток от деления года на 4. Это значение используется в main() только как параметр isLeap(), а потому имеет смысл скомбинировать вычисление remainder и включить его в ту же функцию, в данном случае — в isLeap().

В листинге 8.5 демонстрируется версия программы, в которой обязанности вычисления remainder перенесены из функции main() в серверную функцию isLeap(). Поток данных между функциями показан на рис. 8.5.

Действительно, теперь isLeap() получает у main() только одно значение и вычисляет remainder сама, не заставляя клиента делать это перед вызовом.

Листинг 8.5. Пример переноса обязанностей из клиента в функцию-сервер

```
#include <iostream>
using namespace std;

bool isLeap(int year)                          // еще меньше параметров
{ int remainder=year%4;                        // не следует разделять то, что должно быть вместе
  if (remainder != 0)
     return false;
  else if (year%100 == 0 && year%400 !=0)
     return false;
  else
     return true; }

int main()
{ int year;                                    // локальная переменная - нет remainder
  cout << "Введите год: ";
  cin >> year;                                 // присваивание входных переменных
```

```
if (isLeap(year))
  cout << year << " високосный год\n";
else
  cout << year << " не високосный год\n";
return 0;
}
```

Перенос вычисления остатка из одной функции в другую является перепроектированием: при этом изменяется распределение обязанностей между функциями. Обратите внимание, что здесь объединены ранее разделенные действия. В данном примере обязанности перенесены на сервер, что не всегда дает выигрыш, но часто оказывается полезным.

Это очень мощная техника. Уменьшение коммуникаций между функциями упрощает сопровождение программы, способствует повторному использованию функций и сводит к минимуму необходимые коммуникации между программистами, если функции пишут разные люди (или один и тот же человек в разное время). Каждый раз следует проверять, не разделены ли те части кода, которые должны быть вместе.

Кроме того, не нужно забывать об опасности избыточных коммуникаций между функциями. Лучший способ уменьшить связность — исключить необходимость коммуникаций, совместив те части, которые должны комбинироваться вместе.

Как далеко стоит при этом заходить? Имеет ли смысл переносить в `isLeap()` определение переменной year и вывод запроса пользователю? Это еще более уменьшит поток данных между функциями, однако программистам потребуется согласовывать пользовательский интерфейс (какая функция за какую часть интерфейса отвечает), что проявится в уменьшении сцепления функции `isLeap()`: вычисления в ней будут скомбинированы с вводом-выводом.

В листинге 8.5 функция `main()` отвечает за пользовательский интерфейс, а функция `isLeap()` — за вычисления. Разделение интерфейса с пользователем будет столь же нежелательно, как разделение вычислений. Обязанности каждой функции должны быть четко определены.

Дальнейшие усовершенствования приведенного примера могут включать в себя устранение переменной `remainder` в соответствии с тем, о чем уже говорилось в главе 4.

```
bool isLeap(int year)
  { if (year % 4 || year%100==0 && year%400!=0)
       return false;
    else
       return true; }
```

Те, кто предпочитает компактный код, могут реализовать это таким образом:

```
bool isLeap(int year)
  { return (year % 4 || year%100==0 && year%400) }
```

Как уже говорилось в главе 4, не факт, что эти усовершенствования стоят затраченных усилий, но в любом случае они не влияют на связность, поскольку не изменяют распределения обязанностей между функциями.



Осторожно! Часто степень связности увеличивается, когда разработчики включают в разные функции операции, которые должны реализовываться в одной функции. При этом увеличиваются коммуникации между разработчиками, затрудняется повторное использование функций и сопровождение программы. О такой опасности нужно помнить постоянно.

Инкапсуляция данных

Как и в других языках, в C++ программисты скрывают сложность компьютерных алгоритмов в функциях. Каждая функция представляет собой набор операторов, предназначенных для достижения конкретной цели. Имя функции обычно отражает эту цель. Как правило, имя функции составляется из двух компонентов: глагола, описывающего действие, и существительного, описывающего объект (или субъект) действия (например, `processTransaction()`). Когда объект действия ясен из контекста, (например, когда он передается функции как параметр), можно использовать только глагол (`add()`, `delete()` и т. д.).

Набор операторов в функции может содержать простые операции присваивания, сложные управляющие конструкции или вызовы других функций. Эти другие функции могут быть библиотечными или определяемыми программистом функциями, созданными для конкретного проекта.

С точки зрения программиста, разница между двумя видами функций состоит в том, что реализацию исходного кода функций, разработанных программистами, можно проверить, а исходный код библиотечных функций — нет. Даже когда исходный код библиотечных функций доступен, программист, занимающийся клиентом, не захочет тратить время на их изучение. Ему нужно лишь описание интерфейса серверной функции: какие выходные значения соответствуют входным значениям, какие значения вычисляет функция, какие применимы ограничения и исключения. Это позволяет программисту выбрать соответствующую библиотечную функцию и корректно ее использовать.

Определяемые программистом функции обычно разрабатываются, а не выбираются. Исходный код этих функций часто модифицируется, чтобы он лучше подходил под требования клиентских функций. Эти функции не протестированы так хорошо, как библиотечные функции. Когда возникает проблема, ее источником может быть функция-клиент или любая из серверных функций. Следовательно, программист, занимающийся функцией-клиентом (или сопровождающий ее), должен изучить исходный код связанных с нею функций — клиентов и серверов. Это усложняет задачу по сравнению с использованием библиотечных функций. Желательно разрабатывать функции так, чтобы свести к минимуму подобные дополнительные сложности. Принцип инкапсуляции данных — один из принципов, помогающих программисту достичь данной цели. После успешного тестирования серверных функций они интерпретируются программистами, отвечающими за клиентские и серверные функции, аналогично библиотечным функциям — как "черный ящик" с известным интерфейсом.

Давайте рассмотрим простой пример — часть графического пакета, работающего с геометрическими фигурами, например цилиндрами. Для простоты предположим, что каждый объект-цилиндр характеризуется только двумя значениями типа `double` — радиусом и высотой цилиндра.

```
struct Cylinder {
 .double radius, height; } ;
```

Данная программа запрашивает у пользователя размеры цилиндра. Если объем первого цилиндра меньше объема второго, то она масштабирует первый цилиндр, увеличивая его размеры на 20%, и выводит полученные размеры. В реальной ситуации такой код может быть частью программы, которая использует объекты-цилиндры для описания процессов обмена, происходящих в химическом реакторе, изучения электрического тока в микропроцессоре или анализа стальной фермы. В листинге 8.6 показан пример исходного кода, а на рис. 8.6 — результат выполнения этой программы.

```
Введите радиус и высоту первого цилиндра: 50 40
Введите радиус и высоту второго цилиндра: 70 40

Измененный размер первого цилиндра
радиус: 60 высота: 48
```

Рис. 8.6. *Результат программы из листинга 8.6*

Листинг 8.6. Пример прямого доступа к базовому представлению данных

```
#include <iostream>                                    // пока нет инкапсуляции
using namespace std;

struct Cylinder {                                      // структура данных для доступа
  double radius, height; } ;

int main()
{
  Cylinder c1, c2;
  cout << "Введите радиус и высоту первого цилиндра: ";
  cin >> c1.radius >> c1.height;                       // инициализировать первый цилиндр
  cout << "Введите радиус и высоту второго цилиндра: ";
  cin >> c2.radius >> c2.height;                       // инициализировать второй цилиндр
  if (c1.height*c1.radius*c1.radius*3.141593           // сравнение объемов
    < c2.height*c2.radius*c2.radius*3.141593)
  { c1.radius *= 1.2; c1.height *= 1.2;                // масштабирование
    cout << "\nИзмененный размер первого цилиндра\n";  // вывод нового размера
    cout << "радиус: " << c1.radius << " высота: " << c1.height << endl; }
  else                                                 // в противном случае ничего не делать
    cout << "\nРазмер первого цилиндра не изменен" << endl;
  return 0;
  }
```

Здесь функция `main()` обращается непосредственно к представлению данных `Cylinder` и не прибегает для этого к помощи серверных функций. Она совмещает доступ к данным (например, `c1.radius`) с операциями с данными (такими, как вычисление объема, масштабирование размера или печать данных). В результате сопровождающий приложение программист должен уяснять смысл операций не по именам функций-серверов, а по исходному коду (что сложнее).

Конечно, разработчик программы может написать комментарии и пояснить смысл операторов, как это сделано в листинге 8.6, однако комментарии не всегда достаточно понятны для читателя и не всегда точны. Иногда их просто опускают. Еще хуже, когда у разработчика нет времени на обновление комментариев при изменении исходного кода.

Решение данной проблемы в том, чтобы создать набор серверных функций, обращающихся к полям структуры `Cylinder` от имени клиентского кода. Перекладывая обязанности выполнения вычислений на функции-серверы, можно "очистить" клиентский код от низкоуровневых операций. При этом имена функций позволяют уяснить смысл вычислений "на высоком уровне". В результате исходный код клиента становится понятнее и говорит сам за себя. Читатель понимает, что происходит, по функции-клиенту, даже если не вполне понятно, как именно это делает функция-сервер.

```
int main()                                  // перенос обязанностей на сервер
{
  Cylinder c1, c2                           // данные программы
  enterData(c1,"первого");                  // инициализация первого цилиндра
  enterData(c2,"второго");                  // инициализация второго цилиндра
  if (getVolume(c1) < getVolume(c2))        // сравнение объемов
    {  scaleCylinder(c1,1.2);               // масштабирование
       printCylinder(c1); }                 // вывод нового размера
  else                                      // в противном случае ничего не делать
       cout << "Нет изменений в размере первого цилиндра" << endl;
  return 0;
}
```

Чтобы понять смысл данной версии функции main(), на самом деле нет необходимости разбираться в том, как серверные функции enterData(), getVolume(), scaleCylinder() и printCylinder() делают свою работу. Комментарии здесь те же, что и в листинге 8.6, где не используются функции доступа, но они здесь совсем не помогают, а просто повторяют то, что и так ясно из имен функций, вызываемых клиентом. Это одно из важных преимуществ "переноса обязанностей" с клиента на функции-серверы — принципа, о котором говорилось в начале главы.

При традиционном подходе к программированию строки комментариев очень важны. Если исходный код не содержит комментариев, то программисту следует вернуться назад и добавить их. При инкапсуляции, когда детали вычислений переносятся на серверные функции, клиентский код не нуждается в комментариях. Смысл обработки ясен из имен вызываемых функций-серверов. Если без комментариев код клиента остается не вполне понятным, это означает, что функции-серверы спроектированы не очень хорошо. Программисту следует перепроектировать код (не добавляя комментариев).

Еще одна проблема со стилем программирования в том, что комбинирование доступа к данным с вычислениями их значений затрудняет и делает не очень понятной проверку данных. Часто ее просто опускают. Например, в первой версии программы (листинг 8.6) никакой проверки данных нет. В этом примере данные поступают от пользователя, и следует защитить программу от ошибок. В реальной ситуации данные могут считываться из внешнего файла или поступать по коммуникационной линии. Как и пользователь, эти источники нередко дают запорченные данные. Между тем даже простейшая защита от ошибок (например, присваивание полям Cylinder значений по умолчанию) усложняет код клиента:

```
int main()
{ Cylinder c1, c2;
  cout << "Введите радиус и высоту первого цилиндра: ";
  cin >> c1.radius >> c1.height;                    // инициализировать
                                                    // первый цилиндр
  if (c1.radius < 0) c1.radius = 10;                // по умолчанию на случай
                                                    // порчи данных
  if (c1.height < 0) c1.height = 20;
  cout << "Введите радиус и высоту второго цилиндра: ";
  cin >> c2.radius >> c2.height;                    // инициализировать
                                                    // второй цилиндр
  if (c2.radius < 0) c2.radius = 10;                // по умолчанию на случай
                                                    // порчи данных
  if (c2.height < 0) c2.height = 20;
  if (c1.height*c1.radius*c1.radius*3.141593        // сравнить объемы
      < c2.height*c2.radius*c2.radius*3.141593)
  { c1.radius *= 1.2; c1.height *= 1.2;             // масштабирование
    cout << "\nИзмененный размер первого цилиндра\n";
                                                    // вывод нового размера
    cout << "радиус: " << c1.radius << " высота: " << c1.height << endl; }
  else
    cout << "\nРазмер первого цилиндра не изменен" << endl;
  return 0;
}
```

Использование функций доступа дает возможность устранить проверку данных в клиентском коде на нижнем уровне. Это нетрудно сделать, например, с помощью функции validateCylinder(), устанавливающей поля цилиндра в значения по умолчанию, если введены отрицательные числа. Данная версия программы показана в листинге 8.7. Результат ее будет таким же, как у версии из листинга 8.6.

Листинг 8.7. Пример использования функций доступа для изолирования клиента от имен полей данных

```
#include <iostream>                                    // инкапсуляция в серверных функциях
using namespace std;

struct Cylinder {                                      // структура данных для доступа
     double radius, height; } ;

void enterData(Cylinder &c, char number[])
{  cout << "Введите радиус и высоту ";
   cout << number << " цилиндра: ";
   cin >> c.radius >> c.height; }                      // инициализация цилиндра

void validateCylinder(Cylinder c)
{  if (c.radius < 0) c.radius = 10;                    // значения по умолчанию для данных
   if (c.height < 0) c.height = 20; }

double getVolume(const Cylinder &c)                    // вычислить объемы
{ return c.height * c.radius * c.radius * 3.141593;}

void scaleCylinder(Cylinder &c, double factor)
{ c.radius *= factor; c.height *= factor; }            // масштабирование размеров

void printCylinder(const Cylinder &c)                  // печать состояния объекта
{ cout << "радиус: " << c.radius << " высота: " << c.height << endl;}

int main()
{
   Cylinder c1, c2;                                    // данные программы
   enterData(c1, "первого");                           // инициализация первого цилиндра
   validateCylinder(c1);                               // по умолчанию на случай порчи данных
   enterData(c2, "второго");                           // инициализация второго цилиндра
   validateCylinder(c2);                               // по умолчанию на случай порчи данных
   if (getVolume(c1) < getVolume(c2))                  // сравнить объемы
     { scaleCylinder(c1,1.2);                          // масштабировать
       cout << "\nИзмененный размер первого цилиндра\n";  // вывод нового размера
     printCylinder(c1); }
   else
     cout << "\nРазмер первого цилиндра не изменен" << endl;
   return 0;
}
```

Как можно видеть, данный метод программирования действительно дает более понятный исходный код. В то же время в системах реального времени дополнительные вызовы функций могут повлиять на производительность. Применение встраиваемых функций устранит данную проблему.

Преимущество такого подхода в том, что образуются две разные области: одна относится к проектированию определяемого программистом типа `Cylinder` и его доступа к функциям, а другая — к клиентскому коду, который использует объекты `Cylinder` и вызывает функции доступа `Cylinder`. При традиционном программировании (как в листинге 8.6) таких разделенных областей нет. Если имена полей определенной программистом структуры типа `Cylinder` изменяются, то придется проверять весь код, т. к. эти имена могут использоваться в любом месте программы. В новом варианте (как в листинге 8.7) изменение в именах полей `Cylinder` повлияет только на функции доступа — хорошо определенный набор функций. Остальная часть программы (а она может быть очень большой) не затрагивается.

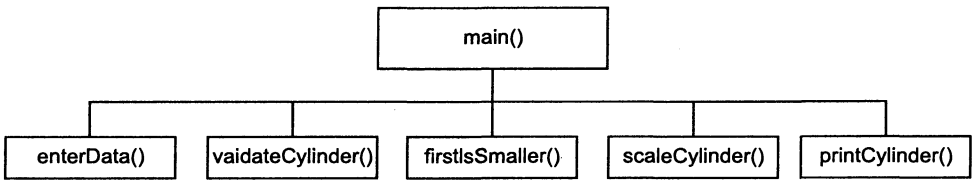


Рис. 8.7. *Структурная диаграмма для программы из листинга 8.7*

Рис. 8.7 иллюстрирует эту взаимосвязь клиентской и серверной части в виде структурной диаграммы. Клиент `main()` вызывает серверные функции, обращающиеся к полям объектов `Cylinder`. Эти серверные функции инкапсулируют функцию-клиент от деталей структуры `Cylinder`.

Инкапсуляция данных — относительно новая концепция, и ее не всегда хорошо понимают. Многие программисты считают, что инкапсуляция данных относится к защите функций от ошибочных и несанкционированных изменений. Без такой инкапсуляции клиентская программа, обращаясь к полям прямо по имени, может произвольно и незаметно изменять данные. При инкапсуляции данных клиент вызывает функции доступа, например `scaleCyliner()`, и эти функции изменяют данные.

Такой вопрос защиты данных аналогичен проблеме использования глобальных переменных. Если глобальные имена доступны во всей программе, то кто-то может некорректно присвоить им значения, что повлияет на другие части программы. Если имена полей данных доступны во всей программе, то может произойти нечто похожее. Передача параметров защищает глобальные переменные, инкапсуляция защищает поля данных.

Эти идеи насчет защиты данных передаются среди программистов из поколения в поколение. Звучит просто и разумно. Легче принять их, чем идти против общего мнения. Здесь нужно возразить. Хотя защита данных действительно играет здесь некую роль, но весьма небольшую. Инкапсуляция данных — прежде всего удобство чтения исходного кода и независимость компонентов программы. Что и составляет основную тему главы.

В действительности передача параметров не защищает переменные. Если кто-то ошибочно думает, что переменной нужно присвоить новое значение, это можно сделать с помощью прямого присваивания (если переменная глобальная) или присваивания значения параметру (если она передается как ссылка или параметр-указатель). Аналогично, если кто-то ошибочно полагает, что полю `c1.radius` следует присвоить новое значение, то это также можно сделать с помощью прямого присваивания (если не используется инкапсуляция) или вызвать функцию доступа, например, `setCylinder()`, когда инкапсуляция применяется. Разницы нет.

Объяснение этого состоит в принципе разделения обязанностей, сформулированном в начале главы. В процессе сопровождения это разделение работ между клиентом и функциями доступа имеет важное значение как для глобальных переменных, так и для полей данных. Если нужно изменить имя глобальной переменной, придется искать ее во всех программных файлах, где она может встречаться, ведь любой файл может обратиться к ней или изменить ее. Четко очерченной сферы полномочий здесь нет — внимание программиста рассеивается по всей программе. Это требует больших трудозатрат и способствует ошибкам.

Аналогично изменение имени или типа поля данных в программе, где данные не инкапсулированы, вынуждает искать все файлы программы, где может встречаться такое поле, поскольку в каждом файле поле может использоваться или модифицироваться. В подобной ситуации также нет ограниченной, небольшой сферы полномочий и нужно заниматься всей программой.

Обратите внимание, о чрезмерном объеме работы при внесении изменений в исходный код речи не идет. В конце концов, сколько времени уходит на его написание и изменение? В проекте разработки это самая простая и короткая часть. Дело в том, что должны быть четко помеченные части программы, где можно проверить изменения при модификации конструкции цилиндра или любых других переменах в структуре данных. Ведь нужно найти все подлежащие изменению места и убедиться, что не внесено никаких побочных эффектов. Вот почему сопровождение программы столь подвержено ошибкам и обходится столь дорого.

В случае инкапсуляции при изменении имени или типа поля данных изменять приходится лишь набор функций доступа. На другие части программы это не влияет. Перекомпилировать также потребуется только те части программы, которые обращаются к данным функциям, но их исходный код при этом не изменяется. Следовательно, сопровождающий приложение программист должен сосредоточить свое внимание на относительно узкой области, ограниченной кодом, имеющим дело с именами полей данных. Вот в чем истинное преимущество инкапсуляции. Если имена полей данных не используются непосредственно, то в клиенте можно обойти зависимость обработки от архитектуры данных.

Очень важно научиться продумывать архитектуру программы с точки зрения инкапсуляции данных. В этом случае создается две разные сферы полномочий: сегменты кода, использующие имена полей данных и не использующие их.

Само по себе применение функций доступа не всегда улучшает читабельность программы и независимость ее фрагментов. Вот почему нужно учитывать еще один критерий, позволяющий судить о качестве программного кода: сокрытие информации.

Сокрытие информации

Принцип сокрытия информации также касается разделения полномочий. Обычно, если намеренное сокрытие информации (о деталях реализации) от пользователя не применяется, программисту, разрабатывающему программу (или сопровождающему ее), приходится помнить одновременно о двух разных областях: архитектуре данных (например, типе `Cylinder`) и операциях с данными на уровне приложения (присваивании значений полям, сравнении объемов, масштабировании размеров и т. д.).

При сокрытии информации обязанности разделяются. Программист, который пишет (или сопровождает) клиентский код, занимается только операциями данных на уровне приложения, а не на уровне архитектуры данных. Программист, отвечающий за функции доступа к данным (или сопровождающий их), занимается лишь архитектурой данных, а не операциями с ними на уровне приложения.

Да, все это звучит похоже на принцип инкапсуляции данных. Нужно признать, что большинство определений сокрытия информации отличается туманностью и неопределенностью. Они не поясняют, как отличить сокрытие информации от инкапсуляции, как распознать недостаточное сокрытие информации и как реализовать такое сокрытие.

Принцип инкапсуляции более узок: он предполагает инкапсуляцию имен и типов полей данных от клиентского кода, чтобы клиент явно не упоминал имен полей данных. В нашем примере будем предполагать, что клиент не должен упоминать `c1.radius`, `c1.height` и т. д. так явно, как в приведенном выше фрагменте. Инкапсуляция через применение функций доступа улучшает качество программного кода, его читабельность и независимость компонентов программы.

Чем сокрытие информации отличается от инкапсуляции? Перед ответом на данный вопрос давайте рассмотрим не очень эффективный пример инкапсуляции. Попробуем реализовать инкапсуляцию, введя функции-серверы, выполняющие операции с объектом `Cylinder`, например возвращающие значения полей `Cylinder`

или вычисляющие объем Cylinder. Эти функции-серверы также называются функциями доступа, так как они обращаются к данным цилиндра от имени клиента. Под "обращением" здесь понимаются вовсе не разные типы доступа, они не различаются. Просто эти функции могут либо считывать поля данных, либо модифицировать их.

```
void setRadius(Cylinder &c, double r)          // функция-модификатор
{ c.radius = r; }

void setHeight(Cylinder &c, double h)          // функция-модификатор
{ c.height = h; }

double getRadius(const Cylinder& c)            // функция-селектор
{ return c.radius; }

double getHeight (const Cylinder& c)           // функция-селектор
{ return c.height; }
```

Функция main() не обязана использовать имена компонентов цилиндра. Если имена изменяются, то изменять придется функции setRadius(), setHeight(), getRadius() и getHeight(), а не main() или других клиентов Cylinder. Пример использования этих функций доступа показан в листинге 8.8. Результат данной программы будет тем же, что и у программы из листинга 8.6. Функциональность ее осталась той же.

Листинг 8.8. Пример неэффективной инкапсуляции

```
#include <iostream>                                    // неуклюжая инкапсуляция
using namespace std;

struct Cylinder {                                      // структура данных для доступа
   double radius, height; } ;

void setRadius(Cylinder &c, double r)                  // модификатор функции
{ c.radius = r; }

void setHeight(Cylinder &c, double h)                  // модификатор функции
{ c.height = h; }

double getRadius(const Cylinder& c)                    // селектор функции
{ return c.radius; }

double getHeight (const Cylinder& c)                   // селектор функции
{ return c.height; }

int main()
{
  Cylinder c1, c2; double radius, height;              // данные программы
  cout << "Введите радиус и высоту первого цилиндра: ";
  cin >> radius >> height;                             // инициализация данных
  setRadius(c1,radius); setHeight(c1,height);
  if (getRadius(c1)<0) setRadius(c1,10);               // проверка данных
  if (getHeight(c1)<0) setHeight(c1,20);
  cout << "Введите радиус и высоту второго цилиндра: ";
  cin >> radius >> height;                             // инициализация данных
  setRadius(c2,radius); setHeight(c2,heihgt);
  if (getRadius(c2)<0) setRadius(c2,10);               // проверка данных
  if (getHeight (c2)<0) setHeight(c2,20);
```

```
    if (getHeight(c1)*getRadius(c1)*getRadius(c1)*3.141593
        < getHeight(c2)*getRadius(c2)*getRadius(c2)*3.141593)
    { setRadius(c1,getRadius(c1)*1.2);
      setHeight(c1,getHeight(c1)*1.2);                    // масштабирование
      cout << "\nИзмененный размер первого цилиндра\n";   // вывод нового размера
      cout << "радиус: "<<c1.radius<<" высота: "<<c1.height<<endl; }
    else
      cout << "Размер первого цилиндра не изменен" << endl;
    return 0;
}
```

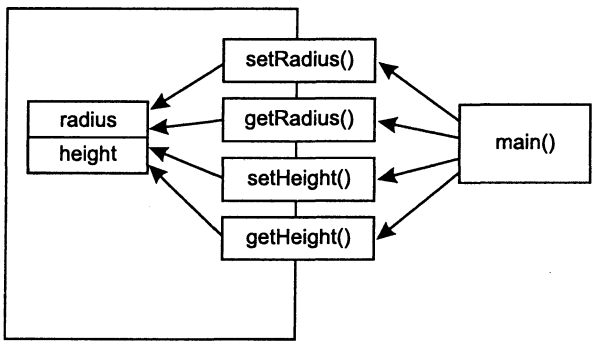


Рис. 8.8. *Диаграмма объектов из листинга 8.8*

Как видно, функция `main()` действительно инкапсулирована от имен полей данных `Cylinder`. Если в ходе перепроектирования эти имена изменятся, то придется модифицировать ограниченный и легко идентифицируемый набор функций. Никаких других частей программы, даже если она очень большая, изменять и даже проверять не нужно. Конечно, придется ее перекомпилировать, но это уже совсем другая история. Рис. 8.8 демонстрирует диаграмму объектов для такой архитектуры. Подобно объектам из диаграммы, приведенной в главе 1 (рис. 1.7), эта диаграмма демонстрирует, что функции-серверы `setRadius()`, `setHeight()`, `getRadius()` и `getHeight()` концептуально родственны. Они обращаются к полям данных структуры `Cylinder` — `height` и `radius` — от лица клиента. Клиент имеет доступ к данным сервера только через вызовы функций доступа, а не непосредственно.

Между тем инкапсуляция здесь достаточно неуклюжая и на самом деле бесполезная. Не используются архитектурные принципы, перечисленные в начале главы. Функции доступа очень мало делают для достижения целей клиентского кода. Ответственность за операции с данными не переносится на функции-серверы, а остается за клиентом. Несмотря на использование функций доступа, в клиенте `main()` смешивается доступ к данным, например, вызовы функций `getRadius()`, с операциями с данными, так что смысл вычислений (вычисление объема, изменение размера) уяснить нелегко. Если изменится число полей определенного программистом типа `Cylinder`, то изменится и число функций доступа, а в результате потребуется модифицировать код клиента.

Для корректного выбора набора серверных функций нужно принимать во внимание обязанности клиента. В данном примере клиент отвечает за инициализацию объектов `Cylinder`, проверку данных объекта, вычисление объема цилиндра, масштабирование его размера и вывод на экран атрибутов цилиндра. Давайте создадим соответствующие функции доступа: `setCylinder()`, `validateCylinder()`, `getVolume()`, `scaleCylinder()` и `printCylinder()`.

Благодаря данным функциям доступа обязанности переносят с клиента на сервер. Именно функции-серверы устанавливают значения полей цилиндра, проверяют данные, вычисляют объем, изменяют размер и выводят результаты. Клиент только запрашивает эти операции. В итоге операции в функции `main()` выражаются в терминах вызовов функций-серверов.

Таким образом, операции с данными уже не смешиваются с доступом к ним. В клиенте определяется, *что* будет сделано (присваивание значений полям, вычисление объема и т. д.), а в серверном коде — *как* это делается. Представление данных `Cylinder` инкапсулировано. Если имена полей изменятся, то на клиента

это не повлияет. Если к Cylinder добавляются поля данных, на клиенте такое изменение не отразится. (На самом деле, это не совсем так, поскольку операции ввода также нужно инкапсулировать.)

Информация, передаваемая от разработчика клиентской части, ограничивается именами и интерфейсами серверных функций. Зоны ответственности разработчиков клиентской и серверной части разделены: одна охватывает связанные с приложением операции высокого уровня, а другая ограничивается именами полей данных и вычислениями нижнего уровня.

Даже в этом небольшом примере видны преимущества применения функций доступа. Код клиента выражается в терминах осмысленных операций уровня приложения. Что означает c1.heihgt*c1.radius*c1.radius*3.141593 в листинге 8.6? Программисту, сопровождающему приложение, придется это выяснить. То же самое относится к операторам c1/radius* = 1.2; и c1.height* = 1.2;. Изменяются ли все размеры цилиндра? Применяется ли ко всем размерам один и тот же коэффициент? Отображают ли операторы вывода все размеры цилиндра или только некоторые? Когда доступ к данным комбинируется с операциями приложения, смысл обработки уяснить труднее.

Применение функций доступа упрощает проверку вводимых пользователем данных — функция main() не перегружается деталями таких операций. Если изменяется представление данных (конструкция цилиндра или просто имена полей), то придется изменять серверные функции. Как уже упоминалось выше, это проблема не только трудозатрат. Вопрос в том, насколько большой области придется уделять внимание. Без функции доступа потенциальной областью изменения является вся программа. (Цилиндры могут использоваться в ней где угодно.) При наличии функций доступа потенциальная область изменений будет четко определена. Она включает в себя функции, обращающиеся к представлению данных цилиндра.

Такой подход упрощает повторное использование программного кода. Без функций доступа любые алгоритмы, работающие с объектами-цилиндрами, придется писать и проверять сначала. Если такие функции имеются, в новых алгоритмах можно вызывать их. Проверять каждую подобную функцию потребуется только один раз.

Недостаток данного подхода состоит в том, что понадобится писать и тестировать больший объем исходного кода, однако можно возразить, что на самом деле это дает дополнительные преимущества. Учитывая общий баланс времени, собственно набор текста (исходного кода) программы составляет лишь малую часть. Все другие шаги разработки — отладка, тестирование, интеграция и сопровождение — требуют чтения исходного кода. Применение при написании клиентской части вызовов функций доступа (уже подготовленных и протестированных) упрощает эти шаги, способствует сокращению числа ошибок и обходится дешевле.

Что же добавляет сокрытие информации к инкапсуляции? Давайте снова рассмотрим серверные функции validateCylinder() и getVolume(). Первая функция инкапсулирует операции проверки данных, значения по умолчанию и т. д. Это хорошо, поскольку клиентскому коду не нужно знать всех деталей проверки допустимости данных. Достаточно, что она выполняется. Вторая функция инкапсулирует геометрические вычисления. Это тоже хорошо, потому что в клиентской программе можно не беспокоиться о правилах геометрии. Достаточно знать, что вычисляется объем цилиндра.

Однако обе эти функции не хороши с точки зрения сокрытия информации. Они расширяют знания разработчика об архитектуре серверной части, увеличивают ту область, которой должен уделять внимание разработчик клиента, и переносят информацию для операций в клиентский код вместо того, чтобы работать с нею на уровне сервера.

Функция validateCylinder() требует проверки данных, в то время как она не должна быть в области внимания разработчика клиента и сопровождающего приложение программиста. Данный недостаток можно устранить с помощью

перепроектирования, т. е. изменения списка функций и их обязанностей. Хорошим решением этой проблемы будет объединение функций validateCylinder() и enterData():

```
void enterDate(Cylinder &c, char number[])
{ cout << "Введите радиус и высоту ";
  cout << number << " цилиндра: ";
  cin >> c.radius >> c.height;                    // инициализировать цилиндр
  if (c.radius < 0) c.radius = 10;                // значения по умолчанию для
                                                  // запорченных данных
  if (c.height < 0) c.height = 20; }
```

Как можно вновь и вновь убеждаться, критерии сцепления, связности, инкапсуляции и сокрытия информации лишь указывают на наличие промахов в архитектуре программы, но не говорят, в каком направлении следует двигаться, как изменить программу, чтобы устранить недостаток. Принципы, перечисленные в начале главы,— это рабочие правила. Они показывают, как нужно изменить архитектуру программы. В данном примере сокрытие информации улучшается путем переноса обязанностей на функции-серверы. Вместо того чтобы вынуждать клиента вызывать две серверных функции, enterData() и validateCylinder(), данная конструкция требует от клиента-вызова только одной функции доступа.

Функция getVolume() нарушает принцип переноса обязанностей на серверные функции, предоставляя клиентской программе больше информации, чем ей нужно. Клиенту необходимо знать только, что один цилиндр больше другого. Вместо обслуживания потребностей клиента серверный код возвращает вычисленное значение объема и позволяет клиенту делать с ним все, что тому заблагорассудится. Информацию об объеме цилиндра следует скрыть от клиентского кода. Для этого надо изменить архитектуру программы, введя, например, функцию firstIsSmaller():

```
bool firstIsSmaller(const cylinder& c1, const Cylinder& c2)
{ if (c1.heihgt*c1.radius*c1.radius*3.141593          // сравнение объемов
      < c2.heihgt*c2.radius*c2.radius*3.141593)
    return true;
  else
    return false; }
```

В листинге 8.9 приведена версия исходного кода, в которой скомбинированы правильная инкапсуляция и сокрытие информации. Обратите внимание, что во всех версиях данной программы функциональность остается одной и той же. Изменяется только архитектура, и именно она влияет на качество программы. Результат программы будет тем же, что и у программы из листинга 8.6.

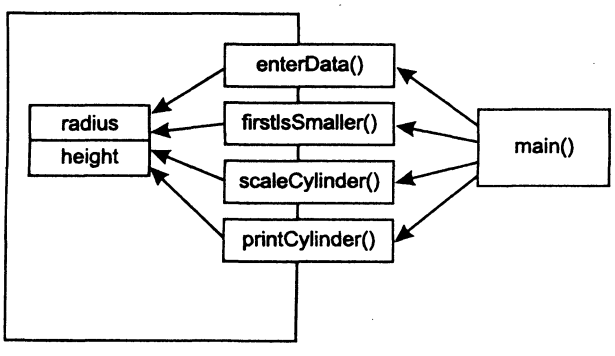


Рис. 8.9. *Диаграмма объектов в листинге 8.9*

На рис. 8.9 показана диаграмма объектов для данной программы. Видно, что, подобно предыдущему рисунку, функции enterData(), firstIsSmaller(), scaleCylinder() и функция printCylinder() родственны (относятся к одной категории). Сервер и клиент оформлены лучше, так как функции доступа выполняют работу для клиента, а не просто передают ему информацию для дальнейших операций.

Листинг 8.9. Комбинирование инкапсуляции и сокрытия информации

```
#include <iostream>
using namespace std;

struct Cylinder {                                          // структура данных для доступа
   double radius, height; } ;

void enterDate(Cylinder &c, char number[])
{ cout << "Введите радиус и высоту ";
  cout << number << " цилиндра: ";
  cin >> c.radius >> c.height;                             // инициализировать цилиндр
  if (c.radius < 0) c.radius = 10;                         // значения по умолчанию
  if (c.height < 0) c.height = 20; }

boolfirstIsSmaller(const cylinder& c1, const Cylinder& c2)
{ if (c1.heihgt*c1.radius*c1.radius*3.141593               // сравнение объемов
      < c2.heihgt*c2.radius*c2.radius*3.141593)
     return true;
  else
     return false; }

void scaleCylinder(Cylinder &c, double factor)
{ c.radius *= factor; c.height *= factor; }                // масштабирование размеров

void printCylinder(const Cylinder &c)                      // вывод состояния объекта
{ cout << "радиус: "<<c.radius <<" высота: "<<c.height<<endl; }

int main()
{
  Cylinder c1, c2;                                         // данные программы
  enterData(c1, "первого");                                // инициализировать первый цилиндр
  enterData(c2, "второго");                                // инициализировать второй цилиндр
  if (firstIsSmaller(c1,c2))
    { scaleCylinder(c1,1.2);                               // масштабирование и
      cout << "\nИзмененный размер первого цилиндра\n";    // вывод нового размера
      printCylinder(c1); }
  else
      cout << "Размер первого цилиндра не изменен" << endl;
  return 0;
}
```

Большой пример инкапсуляции

Следующий пример — это верификация вводимого выражения. Для простоты его функциональность также ограничена — проверяется лишь парность круглых и квадратных скобок во вводимых выражениях. Рассмотрим функцию `checkParen()`, поочередно сканирующую символы выражения в завершающемся нулем массиве (конец выражения) или определяющую несоответствие скобок. Например, такое математическое выражение `a = (x[i] + 5)*y` следует признать допустимым, а выражение `a = (x[i] + 5]*y` — нет.

В данном примере будут использованы два глобальных массива — `buffer[]` и `store[]`. Индекс `i` считывает символ из массива `buffer[]`, а индекс `idx` — из массива `store[]`. Первоначально флаг `valid` устанавливается в 1 (`true`). Если в процессе верификации выражение окажется недопустимым, то названный флаг устанавливается в 0 (`false`). Программа в цикле проверяет следующий символ в массиве `buffer[]`. Если это левая, открывающая скобка (круглая или квадратная), то решение следует отложить до обнаружения парной закрывающей скобки.

Для этого программа сохраняет символ (скобку) в массиве store[] и настраивает индекс idx.

```
char buffer[81]; char store[81];
bool checkParen ()
{ char c, sym; int i, idx; bool valid;
  i = 0; idx = 0; valid = true;              // инициализировать данные
  while (buffer[i] != '\0' && valid)         // конец данных или ошибка?
    {  c = buffer[i];                        // получить следующий символ
       if  (c=='(' || c=='[')                // следующая скобка - закрывающая?
           { store[idx] = c; idx++; }        // затем сохранить ее
// ОСТАЛЬНАЯ ЧАСТЬ ПРОГРАММЫ
  return valid; }
```

Если следующий символ в массиве buffer[] представляет собой закрывающую скобку (круглую или квадратную), то программа считывает последний символ, сохраненный в массиве store[] (опять настраивая индекс idx). При этом если символ в массиве buffer[] является правой круглой скобкой, то символ в массиве store[] должен быть левой круглой (а не квадратной) скобкой. Если два символа не соответствуют (являются парными), то ничего делать не надо — программа переходит к парному символу в массиве buffer[]. Если два символа не соответствуют друг другу, то выражение недопустимо. Программа устанавливает флаг valid в значение false, при этом цикл завершается и клиенту возвращается 0.

```
char buffer[81]; char store[81];
bool checkParen ()
{ char c, sym; int i, idx; bool valid;
  i = 0; idx = 0; valid = 1;                 // инициализировать данные
  while (buffer[i] != '\0' && valid)         // конец данных или ошибка?
    {  c = buffer[i];                        // получить следующий символ
       if  (c=='(' || c=='[')                // следующая скобка - открывающая?
           { store[idx] = c; idx++; }        // затем сохранить ее
       else if (c=='(' || c==']')            // следующая - закрывающая?
       { idx-;  sym = store[idx];            // получить последний символ
         if (!((sym=='(' && c==')') ||
               (sym=='[' && c==']')))        // если непарные
               valid = false; }              // тогда ошибка
// ОСТАЛЬНАЯ ЧАСТЬ ПРОГРАММЫ
  return valid; }
```

Конечно, такой подход слишком оптимистичен. Откуда программа знает, что в массиве store[] всегда есть символ для сравнения с открывающей скобкой из массива buffer[]? Если вводимое выражение содержит несколько правых круглых скобок, не соответствующих предшествующим левым скобкам, то массив store[] будет очищен, его индекс станет отрицательным, а выражение следует объявить недопустимым.

```
char buffer[81]; char store[81];
int checkParen ()
{ char c, sym; int i, idx; bool valid;
  i = 0; idx = 0; valid = 1;                 // инициализировать данные
  while (buffer[i] != '\0' && valid !=0)     // конец данных или ошибка?
  {  c = buffer[i];                          // получить следующий символ
    if (c=='(' || c=='[')                    // следующая скобка - открывающая?
       { store[idx] = c; idx++; }            // затем сохранить ее
```

```
    else if (c=='(' || c==']')           // следующая - закрывающая?
       if  (idx > 0)                     // существует ли сохраненный символ?
           {  idx-; sym = store[idx];    // получить последний символ
              if (!(sym=='(' && c==')') ||
                   (sym=='[' && c==']')))          // если непарные
                 valid = 0; }            // тогда ошибка
       else
           valid = 0;          // если нет парного сохраненного символа, ошибка
// ОСТАЛЬНАЯ ЧАСТЬ ПРОГРАММЫ
  return valid; }
```

Вот почти и все. Мы решили, что нужно делать, если следующий символ в массиве buffer[] является левой скобкой, и что делать в том случае, если это правая скобка. Если же это не правая и не левая скобки, нужно просто перейти к следующему символу в массиве buffer[], т. е. увеличить индекс i.

```
char buffer[81]; char store[81];
bool checkParen ()
{ char c, sym; int i, idx; bool; valid;
  i = 0; idx = 0; valid = true;            // инициализировать данные
  while (buffer[i] != '\0' && valid)       // конец данных или ошибка?
  {  c = buffer[i];                        // получить следующий символ
     if (c=='(' || c=='[')                 // следующая скобка - открывающая?
     { store[idx] = c; idx++; }            // затем сохранить ее
     else if (c=='(' || c==']')            // следующая - закрывающая?
     if (idx > 0)                          // существует ли сохраненный символ?
     {  idx-;    sym = store[idx];         // получить последний символ
        if  (!((sym=='(' && c==')') ||
               (sym=='[' && c==']')))      // если непарные
               valid = false; }            // тогда ошибка
     else
        valid = false;          // если нет парного сохраненного символа, ошибка
i++; }                                    // перейти к следующему символу
// ТО, О ЧЕМ НУЖНО ПОБЕСПОКОИТЬСЯ ПОСЛЕ ЗАВЕРШЕНИЯ ЦИКЛА
return valid; }                          // возврат статуса ошибки
```

В данной программе нужно заботиться еще кое о чем. В конце цикла флаг valid устанавливается в значение false, и это значение должно возвращаться в вызывающую программу. Никаких вопросов не задается — введено недопустимое выражение. Однако если флаг сохраняет значение true, программа тоже должна передать эти "хорошие новости" в вызывающую программу. Во-первых, ей следует проверить наличие дополнительных символов, не совпавших с правыми скобками в выражении, которые могли остаться в массиве store[]. В этом случае idx > 0, выражение недопустимо, а флаг valid следует установить в значение false.

Функция checkParen() наряду с остальным исходным кодом включена в листинг 8.10. Число операторов if в данном примере достаточно велико. Это означает, что для демонстрации корректности функции ее потребуется вызывать не один раз, что усложняет тестирование. Специальная тестовая функция checkParenTest() вызывает функцию checkParen(), показывает введенное выражение и результат выполнения функции. На рис. 8.10 представлен результат выполнения программы.

```
Выражение a=(x[i]+5)*y;
допустимо
Выражение a=(x[i)+5]*y;
недопустимо
```

Рис. 8.10. *Результат выполнения программы из листинга 8.10*

Подобно предыдущим примерам данной главы, здесь делается попытка инкапсулировать программный код от представления символов в функции checkParen(). Этот алгоритм проверки не зависит

от конкретных символов. Например, если нужно иметь дело с фигурными скобками, то алгоритм будет точно таким же. Между тем, если обрабатываемые приложением выражения должны включать в себя фигурные скобки (или другие парные символы), функцию checkParen() следует изменить (наряду с другими функциями в приложении). Возможно, потребуется другое имя функции, поскольку проверяться будут не только скобки.

Листинг 8.10. Пример прямого доступа к представлению данных

```
#include <iostream>                          // пока что нет инкапсуляции
#include <cstring>
using namespace std;

char buffer[81]; char store[81];             // глобальные данные

bool checkParen ()
{ char c, sym; int i, idx; bool valid;
  i = 0; idx = 0; valid = true;              // инициализировать данные
  while (buffer[i] != '\0' && valid)         // конец данных или ошибка?
  { c = buffer[i];                           // получить следующий символ
    if (c=='(' || c=='[')                    // следующая скобка - открывающая?
       { store[idx] = c; idx++; }            // затем сохранить ее
    else if (c==')' || c==']')               // следующая - закрывающая?
       if (idx > 0)                          // существует ли сохраненный символ?
       { idx-;  sym = store[idx];            // получить последний символ
         if (!((sym=='(' && c==')') ||
               (sym=='[' && c==']')))        // если непарные
            valid = false; }                 // тогда ошибка
       else
         valid = false;                      // если нет парного сохраненного символа, ошибка
    i++; }                                   // перейти к следующему символу
    if (idx > 0) valid = false;              // непарная левая скобка - ошибка
    return valid; }                          // возврат статуса ошибки

void checkParenTest(char expression[])       // тестирующая функция
{  strcpy(buffer,expression);
   cout << "Выражение " << buffer << endl;   // вывод выражения
   if (checkParen())                         // проверить допустимость
     cout << "допустим\n";                   // вывод результата
   else
     cout << "недопустимо\n";
}

int main()                                   // инициализатор тестов
{  checkParenTest("a=(x[i]+5)*y;");          // первый тест: допустимое выражение
   checkParenTest("a=(x[i)+5]*y;");          // второй тест: недопустимое выражение
   return 0;
}
```

Серверные функции должны инкапсулировать детали (вид символов и правила поиска парных символов) от клиента. Вот пример трех функций доступа, которые выполняют эту работу. Индекс в массиве символов buffer[] передается функциям isLeft() и isRight(), возвращающим true или false в зависимости от того, на какой символ указывает индекс. Для функции symbolsMatch() передаются два индекса — в массиве buffer[] и в массиве store[], а функция возвращает true или false в зависимости от того, совпадают указываемые этими индексами символы или нет.

```
bool isLeft (int i)
{ char c = buffer[i];                          // получить символ из буфера
  return (c=='(' || c=='['); }                 // проверить, левая ли это скобка

bool isRight (int i)
{ char c = buffer[i];                          // получить символ из буфера
  return (c==')' || c==']'); }                 // проверить, правая ли это скобка

bool symbolsMatch (int idx, int i)
{ char sym = store[idx], c = buffer[i];        // получить два символа
                                               // для сравнения
  return (sym=='('&&c==')')||(sym=='['&&c==']'); }      // совпадают ли они?
```

В листинге 8.11 показана версия программы, использующая данные функции доступа. Если приложение должно работать с фигурными скобками, то потребуется изменить только функции `isLeft()`, `isRight()` и `symbolsMatch()`. Функции `checkParen()` или другой клиентский код модифицировать не потребуется. Результат данной версии программы будет тем же, что и программы из листинга 8.11.

Листинг 8.11. Пример инкапсуляции с общей информацией

```
#include <iostream>                                    // плохой обмен информацией
#include <cstring>
using namespace std;

char buffer[81]; char store[81];                       // глобальные данные

bool isLeft (int i)
{ char c = buffer[i];                                  // получить символ из буфера
  return (c=='(' || c=='['); }                         // проверить, что это левая скобка

bool isRight (int i)
{ char c = buffer[i];                                  // получить символ из буфера
  return (c==')' || c==']'); }                         // проверить, что это правая скобка

bool symbolsMatch (int idx, int i)
{ char sym = store[idx], c = buffer[i];                // получить два символа для сравнения
  return (sym=='('&&c==')')||(sym=='['&&c==']'); }     // совпадают ли они?

bool checkParen ()
{ char c; int i, idx; bool valid;
   i = 0; idx = 0; valid = true;                       // инициализировать данные
   while (buffer[i] != '\0' && valid)                  // конец данных или ошибка?
   { c = buffer[i];                                    // получить следующий символ
     if (isLeft(i))                                    // следующая скобка - открывающая?
       { store[idx] = c; idx++; }                      // затем сохранить ее
     else if (isRight(i))                              // следующая - закрывающая?
       if (idx > 0)                                    // существует ли сохраненный символ?
       { idx-;                                         // получить последний символ
         if (!(symbolsMatch[idx,i])                    // если непарные
             valid = false; }                          // тогда ошибка
       else
         valid = false;                    // если нет парного сохраненного символа, ошибка
   i++; }                                              // перейти к следующему символу
   if (idx > 0) valid = false;                         // непарная левая скобка - ошибка
   return valid; }                                     // возврат статуса ошибки

void checkParenTest(char expression[])                 // тестирующая функция
{  strcpy(buffer,expression);
   cout << "Выражение " << buffer << endl;             // вывод выражения
```

```
   if (checkParen())                                    // проверить допустимость
      cout << "допустимо\n";                            // вывод результата
   else
      cout << "недопустимо\n";}

int main()                                              // инициализатор тестов
{
   checkParenTest("a=(x[i]+5)*y;");                     // первый тест: допустимое выражение
   checkParenTest("a=(x[i]+5]*y;");                     // второй тест: недопустимое выражение
   return 0;
}
```

Хорошая ли это инкапсуляция? Не очень. Представление символов (скобок) действительно скрыто от клиента, но серверные функции работают не только с представлением символов и правилами сравнения. Совместно с клиентом они оперируют массивами `buffer[]` и `store[]`. Обязанности клиента и функций-серверов плохо разделены. Никакой особой причины для такой работы с массивами нет. Как это обычно бывает, при изменении архитектуры программы потребуется менять обе группы функций. При изменении имен этих массивов или переходе с массивов на связные списки должна меняться только функция `checkParen()`, а этого нет. При такой архитектуре потребуется модифицировать функции доступа. Если глобальные массивы здесь не подходят, то изменится интерфейс функций: массивы потребуется передавать как параметры.

Это достаточно редкая форма нарушения принципа сокрытия информации. Обычно лишнюю информацию проявляет клиентский код. В данном примере такое свойственно и серверам. Серверные функции должны знать только одну структуру данных и скрывать это знание от посторонних.

Чтобы обеспечить качество программы на C++, следует постоянно следить за таким совместным использованием информации. Сейчас мы говорим об этом по другому поводу, но позднее вернемся к данной теме.

Для устранения подобного недостатка попробуем снова перепроектировать программу и изменить распределение обязанностей. Передавая функциям сами символы, а не их индексы, скроем индексы массивов от функций-серверов. В версии, показанной в листинге 8.12, функции доступа к символам `isLeft()`, `isRight()` и `symbolsMatch()` знают только о символах, а не о способе их хранения. О массивах знает только клиент.

Листинг 8.12. Улучшенный пример инкапсуляции

```
#include <iostream>                                     // лучший обмен информацией
#include <cstring>
using namespace std;

bool isLeft (char c)
{ return (c=='(' || c=='['); }                          // проверить, что это левая скобка

bool isRight (char c)
{ return (c==')' || c==']'); }                          // проверить, что это правая скобка

bool symbolsMatch (char c, char sym)
{ return (sym=='('&&c==')')||(sym=='['&&c==']'); }      // совпадают ли они?

bool checkParen (char buffer[])                         // выражение - параметр
{  char store[81];                                      // локальный массив
   char c,sym; int i, idx; bool valid;
   i = 0; idx = 0; valid = true;                        // инициализировать данные
```

```
  while (buffer[i] != '\0' && valid)          // конец данных или ошибка?
  { c = buffer[i];                            // получить следующий символ
    if (isLeft(c))                            // следующая скобка - открывающая?
      { store[idx] = c; idx++; }              // затем сохранить ее
    else if (isRight(c))                      // следующая - закрывающая?
      if (idx > 0)                            // существует ли сохраненный символ?
        { sym = store[-idx];                  // получить последний символ
          if (!(symbolsMatch(c,sym))          // если непарные
              valid = false; }                // тогда ошибка
    else
      valid = false;                  // если нет парного сохраненного символа, ошибка
  i++; }                                      // перейти к следующему символу
  if (idx > 0) valid = false;                 // непарная левая скобка - ошибка
  return valid; }                             // возврат статуса ошибки

void checkParenTest(char expression[])
{ cout << "Выражение " << Expression << endl;  // вывод выражения
  if (checkParen(expression))                 // проверить допустимость
        cout << "допустимо\n";                // вывод результата
  else
        cout << "недопустимо\n";}

int main()
{
  checkParenTest("a=(x[i]+5)*y;");            // первый тест: допустимое выражение
  checkParenTest("a=(x[i)+5]*y;");            // второй тест: недопустимое выражение
  checkParenTest("a=(x(i]+5]*y;");            // третий тест: недопустимое выражение
  return 0;
}
```

В данной версии программы инкапсуляция намного лучше, а разделение обязанностей более согласованно. Клиент знает о массивах и индексах, а серверные функции — о символах и правилах их сопоставления.

Знание об одном из массивов, `buffer[]`, для клиента естественно. Это массив, обрабатываемый `checkParen()`. Его инкапсуляция особого смысла не имеет. Если обработка выражения выполняется поэтапно, то функция `checkParen()` будет одной из функций доступа, осуществляющих проверку допустимости и обработку выражения.

Между тем `checkParen()` использует и другой массив — `store[]`. Этот массив усложняет исходный код. Программист должен решить, инициализировать ли `idx` нулем или каким-то другим значением. Когда символ сохраняется в массиве, программисту приходится сначала решать, нужно ли сохранять первый символ, а затем увеличивать индекс. При считывании символа из массива следует определить, нужно ли сначала получить символ, а потом увеличить индекс, или делать это каким-то другим способом. (Надо отметить, что ответы на два последних вопроса различны.) Кроме того, когда функция `checkParen()` проверяет, остались ли в массиве `store[]` непарные символы, приходится решать, сравнивать ли индекс с нулем, единицей или каким-то другим значением.

Ответить на эти вопросы несложно, так как программа невелика, однако в сочетании с другими вопросами все становится труднее, увеличивается вероятность ошибки на этапе разработки и особенно на этапе сопровождения. Еще важнее, что эти проблемы имеют мало общего с алгоритмом, реализующим `checkParen()` — просмотром символов, сохранением левых скобок и их извлечением при обнаружении правой скобки. Каждая функция должна работать только с одной неинкапсулированной структурой данных, и для `checkParen()` такой структурой является массив `buffer[]`, а не `store[]`.

Вот почему следующим шагом в данном примере должна стать инкапсуляция массива store[] и его индекса idx в отдельной структуре, а также реализация функций доступа, которые сможет использовать функция checkParen() для работы с компонентами данной структуры.

```
struct Store {
  char a[81];                    // массив для временного хранения
  int idx; };                    // индекс первой доступной ячейки

void initStore (Store &s)
  { s.idx = 0; }                 // индекс пустой ячейки

bool isEmpty (const Store& s)
  { return (s.idx == 0); }       // проверка, пуст ли store

void saveSymbol (Store &s, char x)
{ s.a[s.idx] = x;                // сохранить символ в store
  s.idx++; }

char getLast(Store &s)
{ s.idx-;                        // вернуться к последнему сохраненному символу
  return s.a[s.idx]; }
```

Опытные читатели, возможно, распознают эту структуру как общий стек, реализованный с помощью массива фиксированного размера. Если вы не знакомы с подобными структурами данных, не стоит беспокоиться. Важно то, что функции доступа изолируют клиента от всех деталей представления данных и позволяют ему выразить алгоритм в терминах вызовов функций (см. листинг 8.13).

При разработке данного примера сделана попытка сохранить строки комментариев, но, если вернуться к листингу 8.10 (первой версии примера без инкапсуляции), то видно, что там комментарии полезны. Они поясняют смысл операций с данными. Сравните их с версией из листинга 8.13. В этой версии с инкапсуляцией комментарии не нужны. Они просто повторяют то, что видно по исходному коду программы. Смысл операторов выражается в именах вызываемых серверных функций.

Листинг 8.13. Инкапсуляция временного хранения store[]

```
#include <iostream>                    // инкапсуляция с сокрытием информации
#include <cstring>
using namespace std;

struct Store {
  char a[81];                          // массив для временного хранения
  int idx; };                          // индекс первой доступной ячейки

void initStore (Store &s)
  { s.idx = 0; }                       // индекс пустой ячейки

bool isEmpty (const Store& s)
  { return (s.idx == 0); }             // проверка, пуст ли store

void saveSymbol (Store &s, char x)
{ s.a[s.idx++] = x;                    // сохранить символ в store

char getLast(Store &s)
{ return s.a[-s.idx]; }                // вернуться к последнему сохраненному символу

bool isLeft (char c)
{ return (c=='(' || c=='['); }         // проверить, что это левая скобка

bool isRight (char c)
{ return (c==')' || c==']';) }         // проверить, что это правая скобка
```

```
bool symbolsMatch (char c, char sym)
{ return (sym=='('&&c==')')||(sym=='['&&c==']'); }     // совпадают ли они?

bool checkParen (char buffer[])                        // выражение - параметр
{  Store store;                                        // массив инкапсулирован
   char c,sym; int i; bool valid;
   i = 0; initStore(store); void = true;               // инициализировать данные
   while (buffer[i] != '\0' && valid)                  // конец данных или ошибка?
   { c = buffer[i];                                    // получить следующий символ
     if (isLeft(c))                                    // следующая скобка - открывающая?
          { saveSymbol(store,c); }                     // затем сохранить ее
     else if (isRight(c))                              // следующая - закрывающая?
       if (!isEmpty(store))                            // существует ли сохраненный символ?
       { sym = getLast(store);                         // получить последний символ
         if (!(symbolsMatch(c,sym))                    // если непарные
               valid = false; }                        // тогда ошибка
       else
         valid = false;                          // если нет парного сохраненного символа, ошибка
     i++; }                                            // перейти к следующему символу
     if (store.idx>0) valid=false;                     // непарная левая скобка - ошибка
     return valid; }                                   // возврат статуса ошибки

void checkParenTest(char expression[])                 // тестирующая функция
{  cout << "Выражение " << expression << endl;         // вывод выражения
   if (checkParen(expression))                         // проверить допустимость
         cout << "допустимо\n";                       // вывод результата
   else
         cout << "недопустимо\n";}

int main()
{  cout << endl << endl;
   checkParenTest("a=(x[i]+5)*y;");                    // первый тест: допустимое выражение
   checkParenTest("a=(x[i)+5]*y;");                    // второй тест: недопустимое выражение
   checkParenTest("a=(x(i]+5]*y;");                    // третий тест: недопустимое выражение
   cout << endl << endl;
   return 0;
}
```

В данной версии программы отсутствуют детали операций с данными, мешающие понимать смысл действий, отвлекающие внимание разработчика и сопровождающего приложение программиста. Область внимания разделяется на четыре узких зоны, а клиент и сервер не обмениваются знанием о структурах данных. Обязанности доступа к данным переданы функциям-серверам.

Недостатки инкапсуляции с использованием функций

Это превосходный способ разработки ПО. Но в реализации инкапсуляции и сокрытия информации с помощью одних лишь функций есть ряд недостатков. Данные недостатки создатели C++ постарались преодолеть с помощью классов.

Один из недостатков состоит в том, что функции доступа не сообщают программисту то, что задумывал разработчик, а именно — что функции имеют отношение друг к другу и обращаются к одной структуре данных. Лучшее решение состоит в том, чтобы поместить функции `isLeft()`, `isRight()` и `symbolsMatch()` (функции доступа к символам) в один файл, а функции `initStore()`, `isEmpty()`, `saveSymbol()` и `getLast()` (функции, обращающиеся к временному хранилищу) — в другой.

В реальности функции, обращающиеся к одной структуре данных, часто совмещаются с функциями, работающими с другими структурами. Они размещаются в файле по алфавиту, и соотношение между структурами данных и функциями доступа становится неясным. Даже когда родственные функции помещаются в отдельный файл без каких-либо дополнительных функций, такое решение все равно остается "управленческим", а не поддерживаемым языком. В языке C (и в некоторых более ранних) отсутствовал какой-либо механизм указания, что некоторые функции логически связаны друг с другом, но не с другими функциями. C++ предлагает превосходное решение. Он позволяет связывать данные и относящиеся к ним функции доступа в классы (ограничивая всю конструкцию фигурными скобками). Сами границы класса показывают, что функции и данные соотносятся друг с другом и не могут группироваться с другими несвязанными функциями.

Второй недостаток инкапсуляции с помощью функций заключается в ее произвольности. Разработчик клиентской части может использовать функции доступа или отказаться от них, обращаясь непосредственно к полям структуры. Правила языка этого не запрещают. Например, в конце функции `checkParen()` в листинге 8.13 проверяется, осталась ли в `store[]` открывающая скобка, для которой при вызове `checkParen()` не оказалось парной скобки. Для корректности с этой целью нужно было бы использовать функцию `isEmpty()`:

```
if (!isEmpty(store)) valid=false;      // ошибка: непарная открывающая скобка
```

Вместо этого для краткости было использовано имя поля `idx` структуры определенного программистом типа `Store`:

```
if (store.idx>0) valid=false;          // ошибка: непарная открывающая скобка
```

Все преимущества инкапсуляции сведены на нет. Исходный код не говорит сам за себя — смысл нужно уяснять из контекста и комментариев. Задача сопровождающего приложение программиста усложняется необходимостью иметь дело с комбинацией доступа к данным и операций с данными. Если нужно изменить имя поля данных `idx`, например на `top` (более распространенное), потребуется модифицировать и код клиента. Такие зависимости между клиентом и сервером усложняют программу. Именно поэтому не очень хорошо полагаться на благоразумие программиста и считать, что он обойдется с инкапсуляцией данных наилучшим образом. C++ решает проблему, предоставляя разработчику квалификатор доступа `private`, который делает нарушение инкапсуляции невозможным.

Третий недостаток в том, что функции доступа являются глобальными. Их имена — часть глобального пространства имен, отсюда они могут конфликтовать с именами других функций. Следовательно, программисты, работающие над разными частями программы, должны координировать свои действия, чтобы избежать конфликтов имен. Кроме того, это вынуждает программистов разбираться в других частях программы больше, чем необходимо.

C++ разрешает проблему, вводя в дополнение к области действия блока, функции, файла и программы область действия класса. Каждое имя, определенное как компонент класса, находится в области действия этого класса. Тем самым устраняются конфликты имен. Программисту не нужно знать об именах, используемых в других частях программы, если он с ними не работает. Тем самым уменьшается необходимость координации между программистами.

Другим недостатком является требование инициализации многочисленных структур данных в клиенте. Например, переменная `store` в листинге 8.13 инициализируется явным вызовом функции `initStore()`. В результате расширяется область, которой должен уделять внимание сопровождающий приложение программист, создается возможность использования данных без должной инициализации.

В C++ проблема устраняется переносом обязанностей с клиента в специальные серверные функции — конструкторы. Они неявно вызываются при каждом создании объекта класса. В этой функции разработчик класса сервера задает, как должен инициализироваться объект класса. В процессе разделения обязанностей между клиентом и сервером работа передается серверу, за эту инициализацию отвечает программист, занимающийся сервером, а программист, работающий с клиентской частью, от нее освобождается. Кроме того, C++ предусматривает другой тип специальных функций — деструкторы. Они неявно вызываются при уничтожении объекта класса. Эти функции возвращают динамическую память и другие ресурсы, которые получил объект, освобождая от таких действий программиста клиентской части.

В C++ есть еще ряд методов, способствующих связыванию данных и операций, инкапсуляции имен полей сервера, сокрытия архитектуры сервера от клиента, переноса обязанностей с клиентов на серверы, минимизации зависимости между клиентской и серверной частью.

Классы C++ обладают огромным потенциалом повышения качества ПО. Подробнее они будут обсуждены в следующих главах.

Итоги

В данной главе рассмотрено использование функций C++ — основного инструмента программирования. Функциональность программы можно реализовать в C++ многими способами.

Цель переноса обязанностей на функции состоит в том, чтобы получить программу, функции которой понятны и легко сопровождаемы, изолированы от других функций и легко используются в разных контекстах. Все, что требует от разработчика клиента (или сопровождающего его программиста) чтения разных фрагментов в разных частях программы для ее понимания и модификации, препятствует повторному использованию программных компонентов и затрудняет сопровождение.

Критерии читабельности и независимости компонентов слишком общие. Для практики необходимы более конкретные критерии, поддерживаемые конкретной технической частью. В данной главе обсуждались традиционные критерии сцепления и связности, а также объектно-ориентированные критерии — инкапсуляция и сокрытие информации. Рассказывалось также о новых критериях, таких, как перенос обязанностей с клиентских функций на серверные, предотвращение разделения связанных функциональных частей, проблемы разделения и ограничения общей для компонентов информации, а также о том, каким способом (отличным от комментариев) разработчик может сообщить в программе о своих замыслах.

Сцепление описывает, насколько хорошо соотносятся друг с другом элементы функции. Функции с сильным сцеплением делают что-то одно с одним объектом. Функции со слабым сцеплением делают несколько вещей. Избавиться от слабого сцепления можно перепроектированием функции — включением разных операций в разные функции, а не совмещением их в одной. Сцепление — не очень строгий критерий, его следует использовать как дополнение к другим.

Связность описывает интерфейс между функцией-сервером и ее клиентскими функциями. Слабая связность означает, что функции относительно независимы. Самая сильная форма связности состоит в использовании глобальных переменных. Это требует координации между разработчиками, занимающимися клиентскими и серверными функциями. Когда функции применяются в других контекстах, используются те же имена переменных. Чтобы проанализировать поток данных между функциями, придется изучить весь исходный код клиентских и серверных функций.

Если для коммуникаций между функциями применяются параметры, эти функции легче использовать повторно. Разработчикам нужно координировать число и типы параметров, но не их имена. Поток данных можно понять, изучив лишь заголовки функций, а не весь код. Чтобы извлечь преимущества из такого подхода, разработчикам следует помнить рекомендации для передачи параметров, о которых рассказывалось в данной и предыдущих главах.

Для уменьшения связности следует так перераспределить обязанности между функциями, чтобы переместить операции, выполняемые в разных функциях, в одну. Тем самым устранится потребность в дополнительных коммуникациях между функциями. Разработчики всегда должны следить за тем, какие коммуникации действительно необходимы, а каких следует избегать. Это очень важный инструмент в наборе программиста.

Инкапсуляция — метод программирования, изолирующий клиентские функции от имен и полей данных, которые этим клиентам нужны. К таким полям по запросу клиентов обращаются функции-серверы. В клиенте используются вызовы серверных функций, а не обращения к полям структур данных. Тем самым программа становится более сопровождаемой, так как создаются две независимые области. При изменении архитектуры программы изменяются функции доступа, а клиентская часть остается той же. При изменении функций приложения изменяется клиентская часть, а функции доступа сохраняются. Если инкапсуляция не предусматривается, то придется проверять на возможные изменения все фрагменты исходного кода.

Сокрытие информации — это метод программирования, еще более изолирующий функции-клиенты от представления данных. Функции доступа выбираются так, чтобы выполнять операции от имени клиентских функций. Программный код клиента выражается в терминах вызовов серверных функций, имена которых описывают алгоритм клиента. Подобный подход еще более улучшает удобство сопровождения и повторное использование.

Если эти методы применять продуманно и последовательно, клиентский код станет объектно-ориентированным, так как будет выражаться в терминах операций со структурами данных. Однако объектно-ориентированное программирование с использованием функций не решает некоторых вопросов. На уровне языка никак не сказывается, что данные и их функции доступа как-то связаны. При сопровождении программы приходится догадываться об этом, изучая соотношение между функциями и данными по исходному коду. Имена функций доступа являются глобальными в области действия программы, и возможны конфликты имен. Если разработчики клиентской части применяют в клиенте имена полей данных, то преимущества инкапсуляции исчезают.

C++ разрешает эти вопросы за счет ввода в язык конструкций классов. Границы класса показывают, что данные и функции связаны. Каждый класс имеет свою отдельную область действия, и функции доступа с одинаковыми именами в разных классах друг с другом не конфликтуют. Разработчик класса может указать, что данные (и функции) являются закрытыми, и предотвратить доступ к ним из клиентов.

Это впечатляет. Применение классов открывает новые горизонты для разработки высококачественных программ. Начиная со следующей главы, мы займемся классами C++.

Глава 9

Классы C++ как единицы модульности программы

Темы данной главы

- ✔ Базовый синтаксис классов
- ✔ Управление доступом к компонентам классов
- ✔ Инициализация экземпляров объектов
- ✔ Использование возвращаемых объектов в клиенте
- ✔ Подробнее о ключевом слове const
- ✔ Статические компоненты класса
- ✔ Итоги

В предыдущей главе были сформулированы базовые принципы объектно-ориентированного программирования с использованием функций как базовых программных блоков. Применяя при построении программы объектно-ориентированный подход, можно добиться того, что вместо непосредственного вызова и модификации полей данных клиент будет вызывать функции доступа. Серверные функции выполняют операции для достижения целей клиентской части. Обязанности распределяются между функциями так, что клиентские функции не знают о представлении данных, а серверные — об алгоритмах клиента.

В результате создаются независимые области программы. При изменении функций доступа сопровождающему приложение программисту не нужно вводить изменения в функции-клиенты (если не модифицируется интерфейс сервера). При изменении клиентских функций программисту нет необходимости разбираться в деталях обработки данных в функциях-серверах или в терминах операций с данными — они не нуждаются в изменениях. В клиентском коде используются вызовы серверных функций, а не операции с данными. Совмещение тех элементов, которые должны быть вместе (а не разделены), делает функции независимыми друг от друга и также способствует облегчению сопровождения и повторного использования. Диаграммы объектов, нарисованные в предыдущей главе, показывают, что функции-серверы логически связаны друг с другом и с данными, к которым они обращаются.

Отмечалось также, что при применении функций для реализации объектно-ориентированного подхода приходится полагаться на произвол программиста. Серверные функции могут включаться в не относящиеся друг к другу части исходного кода, и разработчик не всегда замечает, что они связаны друг с другом

и с представлением данных. Клиентские функции не обязательно используют инкапсуляцию и могут обращаться к представлению данных непосредственно, создавая тем самым зависимости между разными частями программы.

В условиях нехватки времени или из-за недостаточного внимания программисты могут создать зависимости между функциями. Функции со взаимными зависимостями трудно отслеживать, так как разным программистам, работающим над разными взаимозависимыми функциями, придется координировать свою деятельность, что далеко не всегда удается.

Взаимозависимые функции нередко труднее и сопровождать, так как они требуют от разработчика перед внесением изменений изучения этих зависимостей. Такие функции также труднее повторно использовать, поскольку нельзя перенести в другое приложение лишь одну функцию. Требуется переносить их совместно с данными и другими функциями. Вот почему, чтобы справиться со всеми этими проблемами, программисту требуется всяческая помощь со стороны языка программирования. Для создания более качественных программ C++ предлагает великолепную языковую конструкцию — *класс*. Она физически связывает представление данных и операции (функции) с данными. В противном случае это пришлось бы делать концептуально, в воображении разработчика. Такое связывание данных и операций поддерживает принципы инкапсуляции и сокрытия информации.

В данной главе мы подробнее рассмотрим классы C++. Будет рассказано о синтаксисе и семантике, об определении компонентов класса (данных и функций), пояснено, как определять права доступа к компонентам класса, реализовывать классы в однофайловых и многофайловых программах, определять объекты (экземпляры классов), манипулировать объектами, пересылать сообщения, передавать их как параметры и возвращать из функций.

Мы обсудим также специальные функции-члены класса — конструкторы и деструкторы, которые часто понимают неправильно, а также модификатор `const`, уже описанный в главе 7. Этот модификатор помогает разработчику пояснять свои намерения при проектировании программы, в результате чего ее будет проще сопровождать. Особым видом данных и функций класса являются статические данные и функции. Они позволяют разработчику описывать характеристики, общие для всех объектов класса.

Это весьма серьезная глава. К концу ее вы вполне освоитесь с использованием больших единиц модульности (классов) вместо более мелких единиц (функций). Возможно, количество технических деталей покажется вам чрезмерным. C++ — сложный язык, поэтому нужно время для понимания его концепций, практических подробностей и подводных камней. Тот, кто обещает легкие способы обучения C++, либо лжет, либо сам не понимает сложности данной задачи. Если вы почувствуете, что объем информации слишком велик и вы не все усваиваете, попробуйте поэтапный подход. Пропустите некоторые части данной главы и прочитайте остальные, а потом вернитесь к этой главе для повторного чтения. Попробуйте модифицировать ее примеры, поэкспериментируйте и попытайтесь разными способами выразить на языке C++ одни и те же действия. Вы увидите, насколько прекрасна внутренняя логика соединения разных элементов программирования C++. После этого окажется, что применять язык вовсе не сложно. Но это придет к вам только с практикой.

Чтобы освоить основы использования классов, важно периодически возвращаться назад. Многие программисты "перескакивают" этот этап и слишком быстро переходят к более сложным вопросам, таким, как наследование и полиморфизм, не усвоив предварительно основы. Тем самым они еще больше запутываются и пишут программы, которые трудно понимать, сопровождать и повторно использовать. Кроме того, C++ предлагает лишь набор инструментальных средств. Эти инструменты можно применять неверно (подобно оружию, автомобилям и компьютерам). Одно лишь их использование еще не гарантирует автоматически хороших результатов. Программист должен эффективно употреблять в дело данные инструменты. Удачи.

Базовый синтаксис класса

Целью введения классов в C++ является поддержка практики объектно-ориентированного программирования и устранение недостатков, свойственных применению более мелких единиц модульности — функций.

Первая основная цель введения конструкции класса состоит в связывании данных и операций в один синтаксический блок и обозначении общей принадлежности этих элементов программы. Следующая основная цель заключается в устранении конфликтов имен, благодаря чему данные и функции в разных классах могут без проблем использовать одни и те же имена. Третья важная цель — позволить разработчику сервера управлять доступом к элементам класса извне (из программы-клиента). Четвертой целью является поддержка инкапсуляции, сокрытия информации, переноса обязанностей с клиента на сервер, создание раздельных областей ответственности, устранение необходимости координации действий между программистами, работающими над разными частями программы.

Эти цели — естественное расширение практики применения функций для объектно-ориентированного программирования. Если рассматривать классы C++ как еще одну синтаксическую конструкцию безотносительно к описанным выше целям, то применение классов не улучшит качества программного кода. Убедитесь, что вы уделяете достаточно внимания этим четырем целям и стараетесь достичь их каждый раз, когда включаете в программу очередной класс.

Классы являются основой объектно-ориентированного программирования на C++. Они предоставляют программистам инструменты для создания новых типов данных, более соответствующих поведению реальных объектов, чем типы, реализуемые в программах с функциями. Некоторые эксперты полагают, что центральное место в объектно-ориентированном программировании занимает использование наследования и полиморфизма. С этим трудно согласиться. Есть множество программ, не извлекающих никаких преимуществ от применения наследования и полиморфизма. Между тем каждая большая программа C++ получает преимущества от применения классов, если, конечно, эти классы используются корректно и достигают четырех описанных выше целей. Правильно сконструированная программа C++ представляет собой комбинацию компонентов (модулей), совместно выполняющих общую задачу, но при этом независимых и раздельно сопровождаемых.

Связывание операций и данных

Структуры также поддерживают концепцию связывания, комбинируя поля данных. Они позволяют объединять различные компоненты в сложный объект данных. Такими составными объектами можно манипулировать как единым целым, например передавать функции как параметр или обращаться к компонентам объекта индивидуально.

Но определяя структуру, мы моделируем лишь набор данных, а не их поведение. Разрабатывающий серверную функцию программист сам создает инструментальные средства для работы с данными — определяет набор функций доступа для обращения к данным и операциям с ними от имени функций-клиентов. В "функциональном" или "процедурном" программировании данные и алгоритмы структурно разделяются. Они соотносятся друг с другом только в представлении программиста, а не в программном коде. В примерах, обсуждавшихся в главе 8, для демонстрации общей принадлежности функций и данных использовались диаграммы объектов.

Когда программы состоят из функций, связь между разными функциями в программе не очевидна. Каждая функция может обращаться к любому фрагменту данных в программе. Это еще более затрудняет разработку и особенно сопровождение и повторное использование компонентов программы.

Единственный способ указать, что данные имеют отношение к программному коду, который с ними работает, состоит в том, чтобы поместить элементы данных и прототипы функций в один заголовочный файл или отдельно компилируемый исходный код. Однако файл на диске — это концепция аппаратного обеспечения (или операционной системы), но отнюдь не языка. Вот почему C++ расширяет средство `struct`, позволяя связывать вместе элементы данных класса (содержащие значения) и функции-члены (работающие с этими значениями).

Полученные в результате объекты представляют большие единицы модульности. Программисты, занимающиеся клиентской частью, концентрируют свое внимание на данных и на имеющих к ним отношение функциях, а не на многочисленных автономных функциях, связь которых с данными не очевидна.

В хорошо спроектированной программе C++ доступ к данным класса осуществляется через функции, принадлежащие только этому классу. Клиентский код выражается в терминах операций, а не данных. Это сужает горизонт разработчиков клиентской части и сопровождающих приложение программистов.

Формально включением полей в определение `struct` вы создаете класс C++:

```
struct Cylinder {                    // определенный программистом тип (класс)
  double raduis, height; };          // конец области действия класса
```

В C++ ключевые слова `struct` и `class` — почти синонимы. Для только что созданного класса `Cylinder` можно определить объекты (экземпляры или переменные) этого класса, присвоить значения полям объекта. Такой объект разрешается интерпретировать как единое целое (например, передавать его функции как аргумент или сохранять в файле на диске), либо использовать в вычислениях отдельные его части.

В следующем примере функция `main()` определяет два объекта `Cylinder` (переменные и экземпляры), инициализирует их и сравнивает объемы. Если объем первого объекта `Cylinder` меньше объема второго объекта `Cylinder`, то первый объект `Cylinder` масштабируется на 20% и выводится новый размер первого объекта `Cylinder`. Все это аналогично примеру, уже обсуждавшемуся в начале главы 8.

```
int main()
{ Cylinder c1, c2;                                          // данные программы
  c1.radius = 10; c1.height = 30; c2.radius = 20; c2.height = 30;
  cout << "\Первоначальный размер первого цилиндра\n";
  cout << "радиус: " << c1.radius << " высота: и << c1.height: << endl;
  if (c1.height*c1.radius*c1.radius*3.141593                // сравнить объемы
      < c2.height*c2.radius*c2.radius*3.141593)
  { c1.radius *= 1.2; c1.height *= 1.2;                     // масштабировать
    cout << "\nИзмененный размер первого цилиндра\n";  // вывести нов. размер
    cout << "радиус: " << c1.radius << " высота: " << c1.height << endl; }
  else                                                  // иначе ничего не делать
    cout << "\nРазмер первого цилиндра не изменен" << endl;
  return 0; }
```

В данной программе имена полей данных используются явно. Клиент обращается к значениям полей, когда это необходимо (вычисление объемов, масштабирование размера, печать). Изменения в конструкции `Cylinder` влияют не только на структуру `Cylinder`, но и на программный код клиента. Если сопровождающему приложение программисту потребуется уяснить суть обработки, то нужно будет рассмотреть каждый шаг — вычисление объема, масштабирование, вывод. Чтобы проверить, все ли размеры объекта инициализированы, масштабированы или выведены, нужно ссылаться на определение класса `Cylinder`. Повторное использование этих операций в том же или в другом проекте затруднительно, так как они связаны с контекстом программного кода клиента.

Эти недостатки можно устранить, используя для инкапсуляции операций с полями структуры функции доступа: `setCylinder()`, `printCylinder()`, `getVolume()` и `scaleCylinder()`. В листинге 9.1 показана новая версия клиента и сервера. Результаты программы представлены на рис. 9.1.

Листинг 9.1. Пример использования функций доступа, выполняющих операции от имени клиента

```
#include <iostream>
using namespace std;

struct Cylinder {                                          // структура данных для доступа
  double radius, height; } ;

void setCylinder(Cylinder& c, double r, double h)
{ c.radius = r; c.height = h; }

double getVolume(const Cylinder& c)                        // вычисление объемов
{ return c.height * c.radius * c.radius * 3.141593; }

void scaleCylinder(Cylinder &c, double factor)
{ c.radius *= factor; c.height *= factor; }                // масштабировать размеры

void printCylinder(const Cylinder &c)                      // вывод состояния объекта
{ cout << "радиус: " << c.radius << " высота: " << c.height << endl;
}

int main()                                          // перенос обязанностей на серверные функции
{  Cylinder c1, c2;                                        // данные программы
   setCylinder(c1,20,30); setCylinder(c2,20,30);           // размеры цилиндров
   cout << "\Первоначальный размер первого цилиндра\n";
   printCylinder(c1);
   if (getVolume(c1) < getVolume(c2))                      // сравнить объемы
   {  scaleCylinder(c1,1.2);                               // масштабировать
      cout << "\nИзмененный размер первого цилиндра\n";    // вывести новый размер
      printCylinder(c1); }
   else                                                    // иначе ничего не делать
      cout << "\nРазмер первого цилиндра не изменен" << endl;
   return 0;
}
```

```
Начальный размер первого цилиндра
радиус: 10  высота: 30

Измененный размер первого цилиндра
радиус: 12  высота: 36
```

Рис. 9.1. *Результат программы из листинга 9.1*

Данный пример аналогичен примеру из листинга 8.7, и то, что до сих пор демонстрировалось, не выходит за рамки возможностей обычной структуры. Давайте сделаем еще один шаг: скомбинируем поля данных и функции в один класс. В следующем примере синтаксические границы класса `Cylinder` обозначаются открывающей/закрывающей фигурными скобками и завершающей точкой с запятой.

Этот класс содержит два поля или элемента данных: `radius` и `height`. Кроме элементов данных, класс включает в себя четыре функции-члена (такие функции называются также методами — название пришло из языка SmallTalk и искусственного интеллекта). Функции-члены имеют тот же синтаксис, что и глобальные функции. Они могут использовать параметры и возвращать значения. У каждой функции — своя область действия и локальные переменные. В отличие от глобальных функций, функции-члены определяются внутри границ класса (т. е. его фигурных скобок). Теперь всем видно, что функции `setCylinder()`, `printCylinder()`, `getVolume()` и `scaleCylinder()` связаны и сгруппированы вместе с полями данных `radius` и `height`.

```
struct Cylinder {                                  // начало области действия класса
  double radius, height;                           // компоненты данных класса

void setCylinder(double r, double h)               // функции-члены класса
{radius = r; height = h; }

double getVolume()
{ return height * radius * radius * 3.141593; }          // вычислить объем

void scaleCylinder(double factor)
{radius *= factor; height *= factor; }             // масштабировать размеры

void printCylinder()                               // вывод состояния объекта
{ cout << "радиус: " << radius << " высота: " << height << endl; }
} ;                                                // конец области действия класса
```

Добавление функций-членов не меняет базовых свойств структуры. Это шаблон, который позволяет определять объекты данного класса.

```
Cylinder c1, c2;              // выделяется место для двух экземпляров объектов
```

При обсуждении объектно-ориентированной архитектуры и программ вполне естественно использовать термин "объект". К сожалению, у него может быть разный смысл. Некоторые называют этим термином абстрактную концепцию, важную для приложения, например, могут иметь место объекты транзакций, счетов или заказчиков. Другие обозначают им индивидуальные объекты: счет, принадлежащий конкретному заказчику. И есть такие, кто не вполне знает, что он имеет в виду, употребляя этот термин, но полагает, что другие поймут. Я не настаиваю на том, чтобы вы сразу определились, но, пожалуйста, не попадайте в третью категорию.

В данной книге термины "переменные", "экземпляры", "экземпляры класса", "объекты класса" и "экземпляры объектов" используются как синонимы. Все они обозначают программную сущность, для которой на какой-то период времени при выполнении программы выделяется память (в динамической области или в стеке). Они принадлежат к конкретному классу памяти и подчиняются правилам областей действия. Я стараюсь вовсе избегать термина "объект". Если же возникает необходимость применить его, он употребляется в смысле переменной программы. В большинстве случаев это будет переменная определяемого программистом типа, но вполне можно использовать термин "объект" и для переменных встроенных типов. В программировании этот термин многозначен.

В объектно-ориентированном анализе и проектировании термин "объект" часто применяется к набору потенциальных экземпляров с одними и теми же свойствами. Это применение ближе к концепции класса (определяемого программистом типа), чем к концепции экземпляра объекта. Не будем вдаваться в дискуссию, какое применение наиболее корректно, однако во избежание неоднозначности не следует употреблять термин "объект", не описывая его смысл.

Возможно, высказывания против термина "объект" в книге по объектно-ориентированному программированию звучат забавно, но неразборчивое использование данного термина делает неясным его смысл. Всегда проясняйте, что имеется в виду — один экземпляр в памяти компьютера или обобщенное описание потенциальных экземпляров, когда для создания конкретных экземпляров при выполнении программы будет задействован класс C++.

Наличие функций-членов увеличивает размер определения структуры, но не расширяет размера ее экземпляров. Это все равно будет сумма размеров отдельных полей (с возможным дополнительным местом из-за выравнивания). В данном случае, для каждого экземпляра `Cylinder` выделяется место, достаточное для размещения двух значений `double`.

Исключение конфликтов имен

Открывающая и закрывающая фигурная скобки (и завершающая точка с запятой) образуют область действия класса подобно обычной структуре, формирующей обычную область действия для своих полей. Эта область действия класса, вложенная в область действия файла, аналогична области действия структуры. Разница в том, что класс может содержать вложенные области действия функций.

Функции, не являющиеся членами класса (например, функции доступа из листинга 9.1), представляют собой глобальные функции, и их имена должны быть уникальны в программе (если они не являются статическими в области действия файла). Лишь небольшое число функций можно сделать статическими в каждой области действия. Подробнее о статических функциях рассказано в главе 6 и в данной главе. Обычно имена функций должны знать только те разработчики, которые используют класс Cylinder, поскольку им нужно вызывать эти функции. На практике каждому разработчику следует знать эти имена, во избежание конфликтов имен. Данная информация усложняет коммуникации между программистами.

Когда функции реализуются как функции-члены класса (подобно функциям в листинге 9.2), их имена локальны в области действия класса. В результате нельзя вызывать функции-члены или обращаться к элементам данных класса просто с помощью их имен, например radius или setCylinder(). Нужно указывать, какому экземпляру Cylinder (какой переменной) принадлежат данные radius или функция setCylinder().

```
c1.radius = 10;                    // радиус c1
c2.setCylinder(20,30);             // setCylinder() для c2
```

В данном примере значение 10 получает radius экземпляра Cylinder переменной c1, а для вызова setCylinder() используется переменная c2 экземпляра Cylinder.

Если приложение использует другой класс, например Circle, где также имеется компонент данных radius, то конфликта имен не будет:

```
struct Circle {
  double radius;                   // может быть целым или иным типом
... } ;
```

Для доступа к полю radius класса Circle приложению нужно определить экземпляры объекта Circle и их имена:

```
Circle cir1; cir1.radius = 10;     // нет неоднозначности:
                                   // Circle, а не Cylinder
```

Все компоненты класса (элементы данных и функции-члены) находятся в одной области действия (ограниченной фигурными скобками класса). Следовательно, они могут обращаться друг к другу просто по имени без уточняющих ссылок (операций области действия) с именем класса или именем объекта. Например, функция setCylinder() присваивает значения полям (компонентам-данным) radius и height:

```
void setCylinder(double r, double h)      // присваивает значения полей
{ radius = r; height = h; }
```

К чему здесь относятся radius и height? Это поля некоего объекта Cylinder (экземпляра класса). Когда функция-член, например setCylinder(), вызывается в клиенте, объекту передается сообщение. Клиентская программа (находящаяся вне фигурных скобок класса) идентифицирует получателя сообщения (объект, поля которого используются внутри функции-члена) путем явного применения имени объекта, имени функции-члена и разделяющего их селектора-точки.

```
Cylinder c1, c2;                          // потенциальные получатели сообщений
c1.setCylinder(10,30); c2.setCylinder(20,30);      // сообщения передаются
                                                   // экземплярам c1, c2
```

Сообщения применяются к экземплярам объектов данного класса. Когда передается первое сообщение, внутри setCylinder() используются radius и height объекта c1. Когда передается второе сообщение, внутри setCylinder() используются radius и height (радиус и высота) объекта c2. Это относится не только к изменению значений полей при передаче сообщений, но и к простому их использованию в вычислениях.

```
if (c1.getVolume() < c2.getvolume())          // сравнение объемов
  { c1.scaleCylinder(1.2); . . .              // масштабирование
```

В первом сообщении для определения объема (как бы он ни вычислялся) указываются поля переменной c1. Во втором сообщении применяются поля экземпляра c2. Во всех случаях используется имя объекта, оператор селектора-точки и имя сообщения (функции-члена). Синтаксис сообщения тот же, что и синтаксис для доступа (или изменения) полей структуры. Для обозначения поля можно использовать имя объекта, селектор-точку и имя поля.

```
c1.radius = 40.0; c1.height = 50.0;          // используется переменная c1
```

В листинге 9.2 приведена версия клиента и сервера, в которой применяется класс Cylinder (его поля данных связаны с функциями-членами). Поскольку функциональность этой программы та же, что и программы из листинга 9.1 (изменилась только реализация), результаты программ будут одинаковыми.

Листинг 9.2. Пример связывания данных и функций в классе (одна область действия)

```
#include <iostream>
using namespace std;

struct Cylinder {                                    // начало области действия класса
  double radius, height; ;                           // поля данных для доступа

void setCylinder(double r, double h)                 // присваивание данных Cylinder
{ radius = r; height = h; }

double getVolume()                                   // вычисление объема
{ return height * radius * radius * 3.141593; }

void scaleCylinder(double factor)                    // масштабирование размеров
{ radius *= factor; height *= factor; }

void printCylinder()                                 // вывод состояния объекта
{ cout << "радиус: " << radius << " высота: " << height << endl; }
};                                                   // конец области действия класса

int main()                                  // перенос обязанностей на серверные функции
{  Cylinder c1, c2;                                  // данные программы
   c1.Cylinder(10,30); c2. setCylinder(20,30);       // размеры цилиндров
   cout << "\Первоначальный размер первого цилиндра\n";
   c1.printCylinder();
   if (c1.getVolume() < c2.getVolume())              // сравнить объемы
   {  c1.scaleCylinder(1.2);                         // масштабировать
      cout << "\nИзмененный размер первого цилиндра\n";   // вывести новый размер
      c1.printCylinder(); }
   else                                              // иначе ничего не делать
      cout << "\nРазмер первого цилиндра не изменен" << endl;
   return 0;
}
```

Полезно сравнить листинг 9.2 с листингом 9.1 и проанализировать различия между применением автономных глобальных функций (как в листинге 9.1) и функций, связанных с данными (как в листинге 9.2). При использовании автономных функций доступа объектная переменная, имя которой применяется в функции, передается как параметр:

```
void setCylinder(Cylinder& c, double r, double h)      // функция доступа
{ c.radius = r; c.height = h; }                        // Cylinder - параметр
```

Соответствующий экземпляр объекта следует использовать в вызове как фактический аргумент:

```
setCylinder(c1, 10, 30); setCylinder(c2, 20, 30);
```

Без применения классов было бы в высшей степени некорректно реализовывать функцию setCylinder() без параметра Cylinder и вызывать эту функцию, не передавая ей фактический объект, с которым она должна работать.

```
void setCylinder(double r, double h)              // нонсенс: какой Cylinder?
{ c.radius = r; c.height = h; }
setCylinder(10, 30); setCylinder(20, 30);         // нонсенс: какой Cylinder?
```

При разработке функции как функции-члена класса нет необходимости передавать ей в качестве параметра используемый объект.

```
void setCylinder(double r, double h)     // метод: нет параметра Cylinder!
{ radius = r; height = h; }     // компонентные данные, нет полей-параметров!
```

Вместо этого соответствующий экземпляр объекта задается как получатель сообщения при вызове функции:

```
c1.setCylinder(10,30);                  // объект c1 - получатель сообщения
c2.setCylinder(20,30);                  // объект c2 - получатель сообщения
```

Многие начинающие программисты пытаются комбинировать оба способа. Когда они разрабатывают функции-члены класса, то передают объект, с которым нужно работать, как параметр:

```
void setCylinder(Cylinder& c, double r, double h)        // плохой метод
{ c.radius = r; c.height = h; }

c1.setCylinder(c1,10,30); c2.setCylinder(c2,20,30);      // плохие сообщения
```

Не знаю, что привлекает программистов в этих идиомах C++, но встречаются они довольно часто. Корректна ли такая программа синтаксически? Конечно, иначе программисты не смогли бы ее использовать. А семантически? Да, в противном случае программистам пришлось бы с нею что-то делать. Тем не менее выглядит такая программа неприглядно.

Обратите внимание, что в точности тех же результатов можно было бы достичь с помощью других получателей сообщений.

```
c2.setCylinder(c1,10,30); c1.setCylinder(c2,20,30);      // все равно плохо
```

Похоже, что здесь имеет место передача первого сообщения переменной c2 и второго — переменной c1. Первое сообщение все равно присваивает поля переменной c1, а второе — переменной c2, однако потребуется несколько секунд, чтобы понять, что адресат, получатель сообщения, сам ничего с этим сообщением не делает.

Эта неуклюжая запись — лишь пример, показывающий, как легко в C++ написать программный код, который делает не то, что предполагается, и использует программные компоненты, не имеющие к нему отношения. Такая запись всегда увеличивает сложность программы и связность между клиентом и сервером.

Переход от написания автономных функций к классам требует изменения парадигмы. Вы должны сделать этот шаг и комфортно чувствовать себя при работе и с функциями, и с классами.

Реализация функций-членов вне класса

Нужно отметить, что эти функции-члены корректны только в том случае, если они реализуются в области действия класса, т. е. в его фигурных скобках (как в листинге 9.2). Если реализовать их вне данной области действия, то нужно использовать другой синтаксис. Этот синтаксис применяется, когда соотношение между функциями-членами класса и элементами данных устанавливается через прототипы в объявлении (спецификации) класса, а не с помощью полной реализации, как в листинге 9.2:

```
struct Cylinder {                          // начало области действия класса
  double radius, height;                   // поля данных для доступа
void setCylinder(double r, double h);      // присвоить значения полям Cylinder
double getVolume();                        // вычислить объем
void scaleCylinder(double factor);         // масштабирование размеров
void printCylinder();                      // вывод состояния объекта
} ;                                        // конец области действия класса
```

Это означает, что определения функций (их тела) должны реализовываться отдельно. Обычно спецификация класса содержится в заголовочном файле с расширением `.h`, а реализация функций — в исходном файле с расширениями `.cpp` или `.cxx` (в зависимости от компьютера).

Прототипы функций в объявлении класса выглядят точно так же, как прототипы автономных глобальных функций. Единственное различие в том, что они определяются в области действия класса. Границы класса следует принимать всерьез. Программисты почти никогда не забывают об открывающей и закрывающей фигурных скобках, но забывают о точке с запятой после фигурной скобки. К сожалению, компилятор редко сообщает о том, что отсутствует завершающая точка с запятой, а это распространенная ошибка программирования. Обычно он указывает на ошибку в следующей строке программы. Когда содержащая ошибку строка следует за спецификацией класса, нужно проверить, не пропущена ли точка с запятой.



Осторожно! Программисты иногда забывают о точке с запятой после закрывающей фигурной скобки класса. Часто при пропуске точки с запятой компилятор указывает на следующую строку программы, а не на строку, где отсутствует этот символ.

Когда функции-члены реализуются вне спецификации класса, нужно указать имя класса, к которому принадлежит функция. Это естественно, поскольку класс имеет собственную область действия. Каждый класс может, например, содержать функцию-член `getVolume()`. Теоретически каждый класс может иметь функции `setCylinder()` или `printCylinder()`, но вероятность применения этих имен функций для таких классов, как, например, `Cube`, `Circle` или `Account`, не очень высока. В них программисты скорее всего будут использовать имена вида `setCube()`, `setAccount()`, `printAccount()` и т. д. В то же время функция `getVolume()` может использоваться в классах `Cylinder`, `Cube` или `Circle` (в последнем случае с нулевым результатом).

До появления C++ имена вида `setCylinder()` и `setAccount()` были единственной разумной возможностью, ведь в языке C не различались функции с разными сигнатурами. C++ различает функции с одним именем и разными сигнатурами

(см. главу 7 и раздел о перегрузке имен функций), поэтому в нем можно применять вместо имен setCylinder() и setAccount() имя set(). Первая функция set() может иметь параметр Cylinder, а вторая — Account.

С введением в C++ области действия классов конфликты имен функций-членов перестали быть серьезной проблемой. Следовательно, можно использовать имя set() вместо setCylinder(), print() вместо printCylinder() и т. д.

```
c1.set(10,30); c2.set(20,30);      // объекты Cylinder как получатели сообщений
```

Когда компилятор обрабатывает сообщение, он идентифицирует имя целевого объекта и ищет определение (или описание) данного объекта, чтобы идентифицировать его тип. В этом случае компилятор легко установит, что экземпляры объектов c1 и c2 имеют тип Cylinder. Затем компилятор ищет определение (или описание) данного типа и смотрит, определена ли там функция-член с указанным в сообщении именем. Если он находит компонентную функцию set(), то проверяет ее интерфейс — число и типы аргументов. Если число аргументов совпадает, но типы не соответствуют, то компилятор рассматривает возможность преобразования типов. Когда в результате удается добиться соответствия типов аргументов, компилятор генерирует объектный код. В противном случае он выводит сообщение об ошибке.

Все это хорошо, компилятор без труда установит тип целевого объекта. Он просто выполнит поиск по исходному коду (точнее, по таблицам, создаваемым при обработке описаний переменных). Однако этого нельзя сказать о сопровождающем приложение программисте. Ему придется просматривать исходный код программы, что может потребовать много времени и способствовать ошибкам. С этой точки зрения более длинное имя функции поможет программисту избежать поиска определения адресата сообщения:

```
c1.setCylinder(10,30);          // объекты - Cylinder, не так ли?
c2.setCylinder(20,30);
```

Вернемся к обсуждению реализации функций-членов класса, когда спецификация содержит только прототипы этих функций. Реализация их будет точно такой же, как и функций в границах класса. Единственная разница — в именах функций: они должны содержать уточняющее имя класса. Для этого используется операция области действия (::), разделяющая имя класса и имя функции:

```
inline void Cylinder::setCylinder(double r, double h)
{ radius = r; height = h; }                  // присвоить значения полям данных

inline double Cylinder::getVolume()
{ return height * radius * radius * 3.141593; }  // вычислить объем

inline void Cylinder::scaleCylinder(double factor)
{ radius *= factor; height *= factor; }          // масштабировать размеры

inline void Cylinder::printCylinder()            // вывести состояние объекта
{ cout << "радиус: " << radius << " высота: " << height << endl; }
```

В человеческом понимании реальное имя функции-члена setCylinder() — не просто setCylinder(), а функция setCylinder() класса Cylinder. Синтаксически это обозначается как Cylinder::setCylinder().

Такое определение класса из двух частей (спецификация с прототипами и отдельными реализациями) позволяет получить в точности такой же класс Cylinder. Отметим, что когда функция-член реализуется внутри спецификации класса, она по умолчанию встраиваемая (inline), а при отдельной реализации — нет, поэтому в последнем случае ее нужно явно определять как inline.

Как уже отмечалось, спецификация класса с прототипами обычно помещается в заголовочный файл, а реализации функций — в отдельный исходный файл.

Можно, однако, реализовать все или некоторые функции-члены в заголовочном файле. Этот заголовочный файл должен включаться в каждый файл, где упоминается имя класса, например в исходный файл клиента и даже в исходный файл, где реализуются функции-члены (так как имя класса применяется в операции области действия).

Поскольку компоновщик должен видеть определение функции только один раз, спецификации класса следует ограничивать директивами препроцессора для условной компиляции (см. примеры в главе 2 и 5). Например, заголовочный файл для класса Cylinder может выглядеть так:

```
#ifndef CYLINDER_H                    // обычное обозначение символического имени
#define CYLINDER_H
#include <iostream>
using namespace std;

struct Cylinder {                              // начало области действия класса
double radius, height;                         // поля данных для доступа

void setCylinder(double r, double h)           // присвоить значения полям данных
{ radius = r; height = h; }

double getVolume()                             // вычислить объем
{ return height * radius * radius * 3.141593; }

void scaleCylinder(double factor)              // масштабировать размеры
{ radius *= factor; height *= factor; }

void printCylinder()                           // вывести состояние объекта
{ cout << "радиус: " << radius << " высота: " << height << endl; }
} ;                                            // конец области действия класса
#endif
```

В соответствии с общепринятым соглашением этот исходный код нужно поместить в файл Cylinder.h и использовать имя CYLINDER.H для условной компиляции. Реализация функций-членов в отдельном файле — важный вклад в повышение модульности программы. Логически функции-члены, например Cylinder::setCylinder(), определяются в фигурных скобках класса, независимо от того, находятся они в фигурных скобках класса или нет. Вот почему для доступа к элементам данных radius и height эти функции не нуждаются в квалификаторе (операции ::).

В отдельной реализации квалификатор в имени функции обязателен. Без него функция setCylinder() ссылалась бы на глобальные переменные radius и height, а не на компонентные данные radius и height.

```
inline void setCylinder(double r, double h)       // пропущенная операция
                                                  // области действия
{ radius = r; height = h; }                    // присвоить значения полям данных

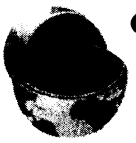
inline double Cylinder::getVolume()            // вычислить объем
{ return height * radius * radius * 3.141593; }

inline void Cylinder::scaleCylinder(double factor)
{ radius *= factor; height *= factor; }        // масштабировать размеры

inline void Cylinder::printCylinder()          // вывести состояние объекта
{ cout << "радиус: " << radius << " высота: " << height << endl; }
```

Компилятор без проблем пропустит такое глобальное определение функции — в C++ это допустимо. Если в области действия файла не описаны переменные radius и height, то компилятор будет жаловаться на то, что они не определены, а вовсе не на отсутствие операции ::. Он даст сообщение об ошибке, вводящее

в заблуждение. Возможно, вы сначала ему не поверите. Неужели компилятор не видит, что radius и height определены вот здесь, в спецификации класса? Наверное это еще одна ошибка в компиляторе! Но компилятор не может знать, что вы забыли применить для определения глобальных переменных операцию ::. Кстати, если переменные с такими именами определены в данной области действия для каких-то других целей, то компилятор будет молча генерировать код, ссылающийся на эти глобальные переменные, а не на элементы данных.



Осторожно! Программисты иногда забывают указывать перед именем функции операцию ::. Компилятор считает при этом, что реализуется глобальная функция, и обвиняет программиста в отсутствии определения глобальных переменных с именами применяемых в функции элементов данных.

Определение объектов классов с разными классами памяти

Область действия класса включает в себя все его данные и функции. Наряду с другими функциями и/или классами они вложены в файл (или в другой класс, функцию, блок), где этот класс объявлен. К компонентам данного класса можно обращаться, когда объект класса находится в области действия.

Что касается переменных любого типа, то объекты класса (его экземпляры, переменные) могут определяться в C++ автоматически (см. главу 6, где рассказывается о классах памяти).

Для автоматических и глобальных (extern или static) переменных память распределяется неявно, в результате определения объекта. Все предыдущие примеры определения экземпляров классов — это примеры автоматических переменных. Они создаются, когда при выполнении программы достигается определение. Например, переменные c1 и c2 в листинге 9.2 создаются, когда выполнение доходит до строки функции main(), содержащей определения этих переменных.

Если переменная класса определяется как глобальная, то память для такого объекта выделяется перед началом выполнения функции main(). Это же относится к случаю, когда переменная класса определяется как статическая (глобальная в файле или локальная в некоторой функции).

Объединяет все такие экземпляры объектов то, что на них можно ссылаться по именам. Для доступа к полям данных и функциям-членам объектов (и ссылки на них) можно использовать имя объекта с точкой. Такой способ применяется в клиенте при обращении к компонентам класса:

```
Cylinder x;
x.setCylinder(50,80);
double volume = x.getVolume(); x.radius = 100;
```

Для динамических переменных память распределяется явно с помощью операции new. Объект не определяется по имени — к нему можно обращаться только через указатель. Для доступа к данным и функциям объекта функция-клиент будет использовать имя указателя (а не имя объекта, так как экземпляры объектов не имеют имен) и обозначение-стрелку. Ниже создается имя указателя на объект Cylinder, а затем — неименованный объект класса Cylinder, с которым выполняются операции:

```
Cylinder* p;                          // объект еще не создан
p = new Cylinder;                     // объект пока не существует
p->set Cylinder(50,80);               // доступ к неименованному объекту
double volume = p->getVolume();       // то же обозначение
p->radius = 100;
```

Если нужно избежать операции стрелки, можно использовать операцию разыменования и селектор-точку:

```
(*p).setCylinder(50,80);                    // то же, что p->set Cylinder(50,80);
```

Аналогично, когда объект передается функции-клиенту по указателю (а не по значению или по ссылке), используется стрелка (а не точка):

```
void CopyData(Cylinder *to, const Cylinder &from)    // копирование Cylinder
{ to->radius=from.radius; to->height=from.height; }  // обозначение-стрелка

Cylinder x,y; x.radius=3.0; x.height=7.0;            // клиент для CopyData()
CopyData(&y,x);                                  // передача объекта по указателю
```

Автоматические переменные уничтожаются, когда завершается область действия, в которой они определены. Программисту не требуется возвращать занимаемую ими память для повторного использования системой. Это же правило действует для глобальных и статических переменных. Они уничтожаются при завершении области действия, т. е. непосредственно после завершения функции `main()`. Никаких программных операций не нужно.

С динамическими переменными все по-другому. Их нужно удалять явно, так как система не знает, когда программист захочет вернуть для повторного использования динамически распределяемую память.

```
Cylinder* p = new Cylinder;                  // создается неименованный объект
p->setCylinder(50,80);                       // доступ к неименованному объекту
cout << "Объем: << p->getVolume() << endl;                  // операция-стрелка
delete p;          // неименованный Cylinder уничтожается, а указатель - нет
```

Как и в случае переменных любого типа, клиенты обращаются к экземплярам класса и их компонентам в соответствии с правилами области действия. Они доступны только тогда, когда экземпляр класса находится в области действия. Кроме того, C++ позволяет разработчику класса устанавливать дополнительные ограничения на другие части программы.

Управление доступом к компонентам класса

В предыдущем разделе был создан класс `Cylinder`, соединяющий свои компонентные данные и функции в один синтаксический блок. Такой синтаксис позволяет решить две проблемы использования глобальных функций для доступа к данным при объектно-ориентированном программировании.

Во-первых, применение глобальных функций для доступа к данным не обязывает указывать, что данные и операции с ними связаны. Следовательно, можно разделить функции, которые соотносятся друг с другом, и разнести их по разным частям исходного кода программы (в результате затрудняется ее понимание при сопровождении и модификации). Во-вторых, имена глобальных функций действительно глобальны. Чтобы избежать потенциальных конфликтов имен, программистам придется координировать свою деятельность, даже когда разрабатываемые ими части программы не связаны друг с другом непосредственно. Синтаксис класса явно показывает, что функции и данные — одно целое, а область действия класса исключает потенциальные конфликты имен функций.

В начале данной главы уже упоминали две цели введения классов в C++: перенос обязанностей с клиента на функции-серверы и управление доступом к компонентам класса.

Перенос обязанностей с клиента на сервер в классе выполняется путем правильного выбора функций-членов. (Иногда важен также выбор элементов данных.) В главе 8 приводился пример (листинг 8.8), где выбор функций-членов

setRadius(), getRadius(), setHeihgt() и getHeight() вынуждал клиента самому выполнять операции, а не обращаться для этого к функциям-серверам. С этой точки зрения выбор функций-членов в листинге 9.2 предпочтительнее. Вместо получения значения radius и height для масштабирования, печати или вычисления объема клиент просит объекты класса Cylinder масштабировать цилиндр и выводить состояние либо вычислять объем.

Перенос обязанностей на функции-серверы — очень важная концепция. Что здесь достаточно, а что нет — часто является субъективным. Можно, конечно, отмахнуться от примера из листинга 8.8, но и он будет полезен, если класс используется как библиотечная утилита, обслуживающая большое число пользователей. Для некоторых пользователей подход, применяемый в листинге 9.2, будет слишком общим. Им может потребоваться вычисление поверхности цилиндра, а не численное значение его объема, но они могут быть заинтересованы и в сравнении объектов-цилиндров (как в листинге 8.9).

О переносе обязанностей на серверные классы часто заходит речь при обсуждении построения классов. В данном разделе рассказывается о некоторых методах, позволяющих разработчику класса управлять доступом к элементам данных и функциям класса.

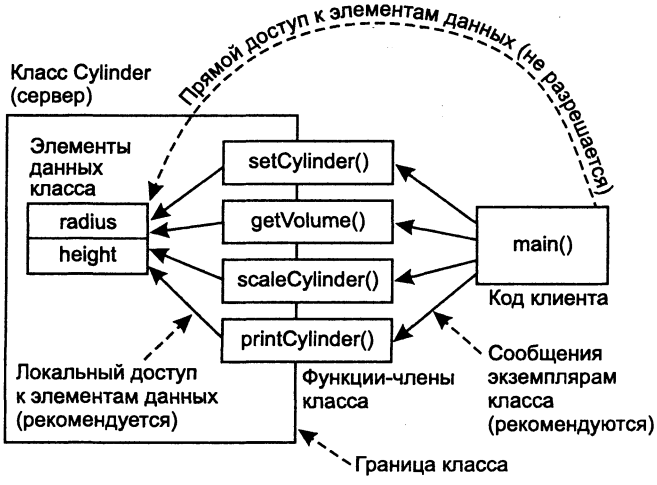


Рис. 9.2. *Класс Cylinder и его взаимосвязь с клиентом main()*

На рис. 9.2 показана взаимосвязь класса Cylinder с клиентом main(). Здесь класс имеет три компонента: данные, функции и границу, отделяющую все, что находится внутри класса, от того, что находится вне его. Он показывает, что данные расположены внутри класса, а функции частично находятся внутри класса (их реализация), а частично — снаружи (интерфейсы, известные клиенту). Этот рисунок демонстрирует также, что когда клиенту нужны значения полей Cylinder (например, для вычисления объема цилиндра, масштабирования, печати или присваивания значений полям), он использует функции-члены getVolume(), scaleCylinder() и т. д., а не обращается к значениям полей radius и height. Вот, что означает пунктирная линия. Она показывает, что прямой доступ к данным исключен.

Две причины побуждают ограничение доступа к элементам данных. Первая — необходимость ограничения масштабов изменений в программе при ее модификации. Если интерфейсы функций-членов остаются теми же (а обычно нетрудно сохранить их при изменении архитектуры данных), то модифицировать нужно реализацию функций-членов, а не программный код клиента. Набор подлежащих изменению функций хорошо определен. Все они перечислены в определении класса, и нет необходимости проверять остальную часть программы.

Вторая причина предотвращения прямого доступа клиента к элементам данных заключается в том, что если клиентский код выражается в терминах вызова функций-членов, а не в терминах детальных вычислений со значениями полей, его проще понять. Кроме того, именно это и предполагает перенос обязанностей на функции-члены, которые выполняют работу для клиента, а не просто считывают и присваивают значения полей подобно функциям getHeight() и setHeight().

Для достижения названных преимуществ все, что находится внутри класса, должно быть закрытым (private), недоступным извне. Тем самым предотвращается создание *зависимостей* от серверных данных класса. Не забывайте, что

в программировании слово "*зависимость*" сродни ругательству. Зависимости между разными частями программы могут означать:

- Необходимость координации работ и тесной кооперации между программистами при разработке программы
- Необходимость изучения и изменения программного кода в процессе сопровождения программы
- Трудности повторного использования программного кода в том же или в аналогичном проекте

Между тем конструкция класса в листинге 9.2 не предусматривает никакой защиты от доступа к данным. Клиент может обращаться к полям экземпляров объектов Cylinder, создавая зависимости от архитектуры данных Cylinder. При этом теряются наиболее важные преимущества использования классов.

```
Cylinder c1, c2;                                  // определение данных программы
c1.setCylinder(10,30); c2.setCylinder(20,30);     // использование функции
                                                  // доступа
c1.radius = 10; c1.height = 20; . . .             // все равно работает!
```

C++ позволяет разработчику детально управлять правами доступа к компонентам класса. С помощью ключевых слов public, private и protected можно задать доступ для каждого компонента (данных или функций). Вот еще одна версия класса Cylinder:

```
struct Cylinder {                                 // начало области действия класса
  private:
    double radius, height;                        // данные являются закрытыми
  public:
    void setCylinder(double r, double h);
    double getVolume();                           // вычисление объема
    void scaleCylinder(double factor);
    void printCylinder();                         // вывод состояния объекта
} ;                                               // конец области действия класса
```

Ключевые слова делят область действия класса на сегменты. Все данные или функции, следующие, например, за ключевым словом private, имеют закрытый режим доступа. В этом примере элементы данных radius и height — закрытые (private), а все функции-члены — общедоступные (public).

Сегментов public, private и protected может быть сколько угодно, и следовать они могут в любом порядке. В приведенном ниже примере элементы данных radius определены как private, две функции — как public, затем элементы данных height — как private и еще две функции — как public:

```
struct Cylinder {                                 // начало области действия класса
  private:
    double radius,
  public:
    void setCylinder(double r, double h);
    double getVolume();                           // вычисление объема
  private:
    double height;
  public:
    void scaleCylinder(double factor);
    void printCylinder();                         // вывод состояния объекта
} ;                                               // конец области действия класса
```

Это предоставляет дополнительную гибкость, но программисты обычно группируют все компоненты класса с одинаковыми правами доступа в один сегмент.

В общем случае компоненты класса (данные или функции) в сегментах public доступны для остальной части программы (как в предыдущих примерах).

Компоненты класса (также данные и функции) в сегментах private доступны только для функций-членов данного класса (и для функций с правами доступа friend, о чем будет рассказано в главе 10). Использование имени закрытого компонента класса вне области действия класса (или функции friend) даст синтаксическую ошибку.

Отметим, что эти правила не запрещают объявлять закрытыми данные и делать функции общедоступными. Однако обычно в C++ элементы данных объявляются как private, а функции-члены — как public.

Компоненты класса в сегментах protected доступны для функций-членов данного класса и функций-членов классов, являющихся их наследниками (прямо или косвенно). Обсуждать наследование пока слишком рано, это намного более широкая тема, чем синтаксис. Мы вернемся к ней позднее.

Функции-клиенты (глобальные или функции-члены других классов) могут обращаться к закрытым элементам данных только через функции в части public (если они имеются).

```
Cylinder c1, c2;                                    // определение данных программы
  c1.setCylinder(10,30); c2.setCylinder(20,30);     // использование функции
                                                    // доступа
//  c1.radius = 10; c1.heihgt = 20;                 // это не синтаксическая ошибка
  if (c1.getVolume() < c2.getVolume())              // еще одна функция доступа
    c1.scaleCylinder(1.2);                          // масштабирование
```

Для поддержки клиентов класса и предотвращения нежелательного доступа разработчик класса должен предусмотреть необходимый доступ к данным. Ведь если клиент будет использовать средства класса, которые он применять не должен, образуется избыточная зависимость. Изменение этих средств повлияет и на клиента. Кроме того, чем больше средств класса объявляются общедоступными, тем больше информации потребуется программисту, создающему и сопровождающему клиента, для продуктивного использования экземпляров класса.

При применении закрытого доступа к элементам данных класса детали реализации класса Cylinder будут скрытыми. Если изменяются имена или поля Cylinder, то на клиента это не влияет (если, конечно, интерфейс Cylinder остается тем же). В программном коде клиента не создаются зависимости от архитектуры данных класса Cylinder. Занимающийся разработкой или сопровождением клиента программист может не изучать ее.

Обычно изменяются именно данные. Вот почему в типичном классе элементы данных объявляются как private, а функции-члены — как public. Тем самым улучшается модифицируемость программы и облегчается повторное использование класса. Заметим, что функции-члены класса (public или private) могут обращаться к элементам данных того же класса (public или private).

Таким образом, любую группу функций, обращающихся к одному и тому же набору данных, следует оформлять как функции-члены класса, а вызовы этих функций использовать в клиенте как сообщения экземплярам класса. Тем самым упрощается повторное использование классов.

Класс изолирован от других частей программы. Его закрытые элементы находятся вне пределов досягаемости других функций (подобно локальным переменным функции или блока).

Это свойство уменьшает необходимость координации между разработчиками ПО и снижает вероятность неверного понимания при таком взаимодействии. В итоге улучшается качество ПО.

Во всех предыдущих примерах для определения класса C++ применялось ключевое слово struct. C++ позволяет также использовать для этой цели ключевое слово class. Вот пример класса Cylinder, где вместо struct применяется ключевое слово class:

```
class Cylinder {                              // начало области действия класса
  private:
    double radius, height;                    // данные являются закрытыми
  public:
    void setCylinder(double r, double h);
    double getVolume();                       // вычисление объема
    void scaleCylinder(double factor);
    void printCylinder();
} ;                                           // конец области действия класса
```

Какая разница между этим определением класса и предыдущим? По существу никакой. Здесь определяется в точности такой же класс с теми же объектами. Различия лишь в ключевых словах struct и class. Одно из различий в том, что ключевое слово имеет в C++ только один смысл и используется только для этой цели (для введения в программе определяемого программистом типа, как это делалось в предыдущих примерах). Еще одно отличие между этими ключевыми словами в назначаемых по умолчанию правах доступа. В struct (и union) доступом по умолчанию будет public. В классе это private. Вот и все.

Применение назначаемых по умолчанию прав доступа позволяет по-разному структурировать последовательность полей данных и функций-членов. Следующую версию можно рассматривать как ответ на возможные обвинения некоторых программистов, которые могут сказать, что если сначала описывать данные, а не функции (как сделано в предыдущих примерах), это не выдерживает критики. Назначение конструкции класса состоит в том, чтобы скрыть данные от клиента, поэтому нехорошо открывать спецификацию класса описанием так называемых "скрытых" данных. Клиент использует общедоступные функции-члены. Следовательно, именно их лучше всего перечислять сначала в спецификации класса.

```
struct Cylinder {                             // некоторые предпочитают перечислять
                                              // сначала функции-члены public
    void setCylinder(double r, double h);
    double getVolume();
    void scaleCylinder(double factor);
    void printCylinder();
  private:
    double radius, height;                    // данные являются закрытыми
} ;                                           // конец области действия класса
```

Другие полагают, что для понимания выполняемых функциями операций важно сначала разобраться в данных, следовательно, нет ничего плохого в том, чтобы описывать в первую очередь данные. Кроме того, сокрытие информации не имеет ничего общего с секретами в стиле КГБ, когда имеется в виду *знание* (или незнание) чего-либо. В программировании сокрытие информации и инкапсуляция состоит в предотвращении *использования* информации. В таком случае, если нужно применить назначенные по умолчанию права доступа, то ключевое слово class лучше, чем struct.

```
class Cylinder {                              // некоторые предпочитают перечислять
                                              // сначала элементы данных
    double radius, height;                    // данные являются закрытыми
```

```
  public:
    void setCylinder(double r, double h);
    double getVolume();
    void scaleCylinder(double factor);
    void printCylinder();
} ;                                    // конец области действия класса
```

Некоторые программисты говорят, что ключевое слово struct хуже, чем class, так как при определении класса с использованием прав доступа по умолчанию, данные не будут защищены от использования клиентом, что вредит инкапсуляции.

```
struct Cylinder {                  // используются права доступа по умолчанию
    double radius, height;         // данные не защищаются от доступа из клиента
  void setCylinder(double r, double h);              // методы общедоступные
  double getVolume();
  void scaleCylinder(double factor);
  void printCylinder();
} ;                                // конец области действия класса
```

При такой конструкции класса проигрывает инкапсуляция, но это же не доказывает, что ключевое слово struct хуже, чем class! Если здесь заменить struct на class, то результат будет еще хуже. Видите, почему?

```
class Cylinder {                   // используются права доступа по умолчанию
    double radius, height;         // данные защищаются от доступа из клиента
  void setCylinder(double r, double h);              // методы недоступны
  double getVolume();
  void scaleCylinder(double factor);
  void printCylinder();
} ;                                // конец области действия класса
```

Такой класс вообще нельзя использовать. Да, поля данных теперь закрыты (и это превосходно), но недоступны и функции-члены, и клиент не может вызывать их. Определенно, не очень хорошая конструкция.

Вероятно, предпочтительнее не полагаться на права по умолчанию и назначать права доступа явно. Так что будем называть вещи своими именами.

Инициализация экземпляров объекта

Когда компилятор обрабатывает определение переменной, он использует для выделения требуемого объема памяти определение типа. Память выделяется из динамически распределяемой области (для переменных static и extern или для динамических переменных) или из стека (для локальных автоматических переменных).

Это относится к простым переменным, массивам, структурам и классам с функциями-членами. Если позднее программа присваивает переменной значение, то она не нуждается в инициализации (как в случае определения), но когда алгоритм использует переменную как r-значение, элементам данных необходимы начальные значения.

```
Cylinder c1;                       // элементы данных не инициализируются
double vol = c1.getVolume();       // нет, нехорошо
```

Такой прием программирования может не подойти, если в вычислениях нужно использовать некоторые значения по умолчанию. C++ инициализирует только статические и глобальные переменные (нулями соответствующего типа). Динамические и автоматические переменные остаются без начальных значений.

Иногда желательно определить значения по умолчанию. Хорошо было бы инициализировать элементы данных в определении, подобно обычным переменным, но в C++ определения элементов данных не могут содержать инициализатор.

```
class Cylinder {
  double radius = 100, heihgt = 0; ...          // нет, в C++ это недопустимо
```

Класс может предусматривать функцию-член, позволяющую клиенту задать начальное состояние объекта:

```
class Cylinder {
  double radius, heihgt;
public:
  void setCylinder(double r, double h); . . . } ;
```

С помощью этой функции клиент мог бы передавать сообщение `setCylinder()` объектам `Cylinder`.

```
Cylinder c1;
c1.setCylinder(100.0,0.0);          // присваивает radius значение 100,
                                    // а height ноль
```

Конечно, это перебор. Такой код позволяет задавать любые начальные значения, а не указанные по умолчанию. Здесь становятся полезными конструкторы.

Конструкторы как функции-члены

Объекты класса могут инициализироваться неявно, с помощью конструктора. Конструктор — это функция-член класса, но она имеет более строгий синтаксис, чем другие функции-члены. Конструктор не может иметь произвольное имя. Оно должно соответствовать имени класса. Интерфейс конструктора не может специфицировать возвращаемый тип (даже `void`) и возвращать значения, даже если содержит оператор `return`.

```
class Cylinder {
  double radius, height;
public:
  Cylinder ()               // то же имя, что и у класса, нет возвращаемого типа
    { radius=1.0; heihgt=0.0; }        // нет оператора возврата
  ... } ;
```

Когда клиент создает объект, вызывается конструктор, заданный по умолчанию.

```
Cylinder c1;              // конструктор по умолчанию; нет параметров
```

Он называется конструктором по умолчанию, так как не имеет параметров. Странная причина, но это так.

Конструктор нельзя вызывать явно, как любую другую функцию-член.

```
c1.Cylinder();    // синтаксическая ошибка: явно вызывать конструктор нельзя
```

Конструктор может вызываться только при создании объекта, но не позднее. Компилятор генерирует код, явно вызывающий конструктор сразу после создания экземпляра объекта, поэтому конструкторы обычно включаются в раздел `public` спецификации класса. В противном случае попытка создать экземпляр класса даст ошибку, как любой доступ к частному компоненту класса.

В общем случае экземпляр объекта можно создавать:

- В начале программы (объекты `extern` и `static`)
- На входе в область действия, содержащую определение объекта (автоматические объекты)

- Когда переменная передается функции (или возвращается из функции) по значению
- Когда переменная создается динамически с помощью операции new (но не malloc)

Теперь видно, почему конструкторы не возвращают значений. Они вызываются неявно сгенерированным компилятором программным кодом, и никто не сможет использовать возвращаемое значение.

Как и функции-члены, конструкторы могут иметь параметры. Следовательно, для них возможна перегрузка имен. Если необходимо, параметрам конструктора можно присваивать значения по умолчанию. Когда класс имеет более одного конструктора, при создании объекта может вызываться любой из них. Какой именно конструктор вызывается, зависит от контекста, т. е. набора аргументов, передаваемого клиентом при создании объекта (учитывается число аргументов и их типы).

```
class Cylinder {
  double raduis, heihgt;                    // инициализируются в конструкторах
public:
  Cylinder(double r, double h);             // прототип компонентной функции
  void setCylinder(double r, double h);
  . . . . } ;

Cylinder::Cylinder(double r, double h)   // операция области действия
  { radius = r; heihgt = r; }
```

Обратите внимание на имя конструктора, реализованного вне границ класса. Первый Cylinder обозначает класс, к которому принадлежит функция. Второй Cylinder обозначает имя функции-члена (совпадающее с именем класса). Конструкторы, у которых менее двух параметров, имеют специальные имена. (О них будет кратко рассказано ниже.) Конструкторы с двумя или более параметрами специальных имен не имеют. Это просто общие конструкторы.

Рассматриваемый конструктор с двумя параметрами делает то же самое, что setCylinder() — присваивает значения элементов данных значениям поставляемых клиентом аргументов. Разница в том, что в клиенте setCylinder() может вызываться для одного и того же экземпляра объекта много раз. Конструктор же вызывается только один раз — при создании объекта.

Вот несколько примеров вызова конструктора в клиенте. Эти разные синтаксические формы вызывают один и тот же общий конструктор с двумя параметрами. Обратите внимание, что операция присваивания во втором параметре не означает выполнения присваивания. Несмотря на кажущееся использование присваивания, это не то, что вы думаете (что весьма типично для C++). Здесь вовсе нет никакого присваивания. Просто так выглядит еще одна форма вызова конструктора.

```
Cylinder c1(3.0,5.0);          // вызов конструктора для именованного объекта
Cylinder c2 = Cylinder(3,5);   // все равно вызов конструктора
Cylinder *r = new Cylinder(3.0,5.0);                  // неименованный объект
```

Обратите внимание на синтаксис переменной с аргументами. Это новый синтаксис. Одна из неявных целей, поставленных при разработке C++, состоит в единообразной интерпретации переменных встроенных и определяемых программистом типов. Для встроенных типов мы использовали при инициализации операцию присваивания. С появлением определяемых программистом типов стало возможным применять для переменных синтаксис с аргументами. Это могут быть переменные встроенного типа или объекты классов.

```
int x(20);                     // то же, что x1=20
```

Когда память для объекта выделяется с помощью вызова malloc(), конструктор не вызывается. Следовательно, клиенту нужно явно инициализировать объекты класса.

```
Cylinder p* = (Cylinder*)malloc(sizeof(Cylinder));
                                        // нет вызова конструктора
p->setCylinder(3,5);                    // полям объекта присваиваются значения
```

Вызов malloc() — единственный способ в C++, позволяющий создать объект без помощи вызова конструктора. Создание других объектов (именованных и динамических) сопровождается вызовом конструктора. Итак, мы незаметно миновали точку возврата. Теперь невозможна ситуация, когда вы просто создаете экземпляр объекта и выделяете ему область памяти. Любое создание объекта будет сопровождаться вызовом функции-конструктора. Это опять требует определенной смены мышления. Каждый раз, видя создание объекта, нужно напоминать себе, что это означает вызов конструктора. Но какого?

Конструкторы, используемые по умолчанию

Многие классы не нуждаются в конструкторах, так как объекты этих классов не требуют инициализации по умолчанию. Если разработчик не включает в класс никаких конструкторов, то система подставляет для класса конструктор по умолчанию (который просто ничего не делает).

```
class Cylinder {                    // OK, если нет конструкторов/деструкторов
    double radius, height;          // данные защищены от доступа из клиента
public:
  void setCylinder(double r, double h);        // конструкторы доступны
  double getVolume();
  void scaleCylinder(double factor);
  void printCylinder();
} ;                                 // конец области действия класса
```

Все версии класса Cylinder, обсуждавшиеся в предыдущем разделе, используют конструктор, по умолчанию подставляемый системой. О нем специально ничего не упоминалось, чтобы не усложнять дискуссию. Когда клиент создает объект Cylinder, вызывается этот конструктор по умолчанию.

```
Cylinder c1;      // вызывается конструктор по умолчанию, нет инициализации
```

Зачем нужно знать об этом? Весь такой конструктор ничего не делает. Но следует иметь в виду, что если класс определяет конструктор явно (конструктор с параметрами), то конструктор по умолчанию не подставляется.

Это нужно знать, т. к., если разработчик определяет переменные и массивы, нуждающиеся в конструкторе по умолчанию, возникает синтаксическая ошибка.

Данная версия класса Cylinder не имеет определенных программистом конструкторов. Следовательно, система назначает этому классу конструктор по умолчанию (ничего не делающий). При создании переменной c1 был вызван такой конструктор. Откуда это известно? Но ведь какие-то конструкторы должны вызываться. (Создание объекта без вызова конструктора невозможно.) Так какой же конструктор вызывается? Зависит от числа аргументов. Переменная c1 не предусматривает никаких аргументов. Это говорит о том, что вызывается конструктор без аргументов, т. е. конструктор по умолчанию. Предусматривает ли класс конструктор по умолчанию? Нет. Следовательно, конструктор по умолчанию подставляется системой. Он ничего не делает. Все замечательно.

Давайте рассмотрим версию класса Cylinder, который предусматривает общий определенный программистом конструктор. Это означает, что система не будет использовать конструктор по умолчанию.

```
class Cylinder {
    double radius, height;
  public:
    Cylinder(double r, double h)          // этого недостаточно
    { radius = r; height = h; }
    . . . } ;
```

Когда клиент пытается создать объекты Cylinder, возникают проблемы.

```
Cylinder c1(3.0,5.0);                     // OK
Cylinder c2, c[1000];                     // 1001 синтаксическая ошибка
Cylinder *p = new Cylinder;               // одна синтаксическая ошибка
```

Здесь создается 1001 экземпляр объекта Cylinder без указания аргументов. Помните, что создание объекта без вызова конструктора не возможно? Поэтому компилятор пытается сгенерировать 1001 вызов конструктора. Какого именно? Так как аргументов нет, то он вызывает конструктор без аргументов, т. е. конструктор по умолчанию Cylinder::Cylinder(). Но в данной версии класса Cylinder не определен конструктор по умолчанию, а определен общий конструктор. Таким образом система вызывает общий конструктор, а конструктор по умолчанию не использует. Что будет, если клиент вызовет 1001 раз функцию-член класса и создаст 1001 объект Cylinder? Поскольку эту функцию в спецификации класса найти не удается, генерируется сообщение о синтаксической ошибке. Научитесь быстро делать такие логические выводы.

Проблему можно устранить, предложив клиенту конструктор по умолчанию, определяемый программистом. Подобно конструктору, подставляемому по умолчанию системой, этот конструктор может не выполнять никаких операций, либо инициализировать элементы данных объекта, присваивая им какие-то разумные значения.

```
class Cylinder {
    double radius, height;
  public:
    Cylinder ()       // определенный программистом конструктор по умолчанию
    { radius = 100.0; height = 0.0; }        // разумные значения
    Cylinder(double r, double h)             // общий конструктор
    { radius = r; height = h; }
    . . . } ;
```

В клиенте:

```
Cylinder c1(3.0,5.0);                    // OK
Cylinder c2, c[1000];                    // тоже OK
Cylinder *p = new Cylinder;              // нет синтаксической ошибки
```

Отметим еще раз, что создание каждого объекта сопровождается здесь по крайней мере одним вызовом функции. Конструкторы — это встраиваемые функции (inline). Тем не менее они могут влиять на производительность. В C++ не бывает ситуации, когда объект создается без вызова функции.



Внимание Создание объекта в C++ всегда сопровождается вызовом конструктора. Если конструктор в классе не определяется, то за созданием объекта следует вызов конструктора по умолчанию, поставляемого системой. Если в классе определен какой-либо конструктор, система не подставляет конструктор по умолчанию. В этом случае нельзя создавать массивы или "объекты объектов" без аргументов. Система дала, система взяла.

Конструкторы копирования

Одно из важных понятий, составляющих основу философии объектов C++, является то, что классы — это типы. Определение классов в программе расширяет систему встроенных типов C++. Идея в том, чтобы интерпретировать в C++ встроенные типы как объекты, а определяемые программистом типы интерпретировать как встроенные.

Например, можно определять переменные встроенных типов без спецификации их начальных значений. Следовательно, можно делать это и с объектными переменными.

```
int x; Cylinder c1;                    // неинициализированные переменные
```

Синтаксис тот же, но смысл другой. Определение переменной встроенного типа просто выделяет память для этой переменной. Определение переменной созданного программистом типа приводит к выделению памяти для этой переменной и вызову конструктора по умолчанию. Если класс не определяет конструкторы, то определение переменной класса даст синтаксическую ошибку. Чтобы этого не произошло, в классе должен быть определен конструктор по умолчанию. Этот конструктор может не выполнять никаких действий или инициализировать поля объекта какими-то значениями по умолчанию.

Аналогично может потребоваться инициализация отличной от класса переменной встроенного типа другой переменной того же типа. C++ поддерживает подобный синтаксис, позволяющий клиенту инициализировать один объект класса значениями другого объекта того же класса.

```
int x(20); Cylinder c1(50,70);          // объекты создаются и инициализируются
int y=x; Cylinder c2=c1;     // инициализация с помощью существующих объектов
```

Пусть вас не вводят в заблуждение операции присваивания во второй строке. Никакого присваивания здесь нет. Не забывайте, что когда после имени переменной указывается тип, то речь идет об инициализации. Если имя типа отсутствует, а указано только имя переменной, то мы имеем дело с присваиванием. Для чего это знать? Как вы увидите далее, в каждом случае вызываются разные функции.

Какая же функция вызывается в этом случае? Ответ прост. Так как объект создается и инициализируется, здесь вызывается конструктор. Какой именно? Как уже говорилось выше, это зависит от контекста, т. е. числа и типа фактических аргументов, подставляемых при создании объекта.

В этом примере для инициализации объекта `c2` используется один аргумент — объект `c1`. Он имеет тип `Cylinder`. Следовательно, вызываемый конструктор имеет один параметр типа `Cylinder`. Вывод ясен? Такая цепочка рассуждений должна иметь место каждый раз, когда вы анализируете операторы создания объекта.

Конструктор с одним параметром того же типа, что и класс, имеет специальное название — конструктор копирования. Он называется так потому, что копирует значения из имеющегося источника в поля только что созданного целевого объекта. Как видно, последняя версия класса `Cylinder` не содержит конструктора с одним параметром типа `Cylinder`. Она имеет общий конструктор с двумя параметрами `double` и конструктор по умолчанию без параметров. Означает ли это, что приведенные операторы ошибочны, подобно ситуации, когда вводилась концепция используемого по умолчанию конструктора? Нет, и это еще одно подтверждение, что изучение C++ — нескучное занятие.

Если конструкторы в классе не определены, C++ предусматривает собственный конструктор по умолчанию. Этот конструктор копирует элементы данных по битам из объекта-источника в целевой объект. В отличие от подставляемого системой конструктора по умолчанию, такой подставляемый системой конструктор копирования не игнорируется, даже когда в классе определены другие конструкторы. Следовательно, всегда нужно учитывать его существование.

Для класса, подобного Cylinder, не имеет особого смысла определять свои определяемые программистами конструкторы копирования. Все, что можно сделать в таком конструкторе,— это скопировать поля radius и height параметра, но в точности то же самое делает конструктор копирования, подставляемый по умолчанию. Конструктор копирования, определяемый программистом, можно включить разве что в целях отладки.

```
class Cylinder {
  double radius, height;
public:
    Cylinder (const Cylinder &c)
    {  radius = c.radius; height = c.height;
       cout << "Конструктор копирования: " << radius << ", "
......}; << height << endl; }
```

Обратите внимание, что параметр должен быть ссылкой заданного типа, а не значением данного типа. Что произойдет, если параметр будет передан конструктору копирования по значению?

```
Cylinder (Cylinder c)                    // некорректный интерфейс конструктора
  {  radius = c.radius; height = c.heingt;
     cout << "Конструктор копирования: " << radius << ", " << height << endl; }
```

При вызове такого конструктора выполняется копирование фактических аргументов — выделяется и инициализируется (значениями полей фактического аргумента) память для переменной Cylinder. Но ведь в C++ не бывает создания объекта без вызова конструктора! Выделение и инициализация памяти для значений полей фактического аргумента означает вызов конструктора копирования для параметров конструктора копирования. Когда вызывается эта вторая версия конструктора копирования, делается копия его фактических аргументов и конструктор вызывается снова. Процесс рекурсивных вызовов продолжается, пока у пользователя не лопнет терпение или у машины не исчерпается память в стеке.

Если у вас нет опыта работы с рекурсией, а это объяснение кажется слишком туманным, просто попробуйте передать конструктору копирования параметр по значению. Второй раз вам не захочется этого делать. Тем не менее стоит предупредить о данном эффекте.



Осторожно! Конструктор копирования имеет один параметр типа класса, которому данный конструктор и принадлежит. Этот параметр необходимо передавать по ссылке const, а не по значению. Передача конструктору копирования параметра по значению приводит к бесконечной последовательности вызовов такого конструктора.

И еще один комментарий по конструктору копирования. Поскольку это вызов функции, можно обращаться к конструктору с помощью стандартного синтаксиса вызова функции общего конструктора.

```
int x = 20; Cylinder c1(50,70);    // объекты создаются и инициализируются
int y=x; Cylinder c2(c1);          // вызов конструктора копирования Cylinder
```

Но C++ хочет одинаково интерпретировать объекты и переменные встроенных типов. Это означает, что синтаксис инициализации в вызове конструктора можно распространить и на встроенные переменные, хотя для переменных данных типов никакие конструкторы вызываться не могут. Такой синтаксис доступен только в C++, но не в языке C.

```
int x(20);              // создается и инициализируется объект
int y(x);               // создается и инициализируется переменная y
```

Еще один общий комментарий по поводу вызова конструктора. Для всех конструкторов, за исключением конструктора по умолчанию, можно использовать синтаксис вызова функции (со скобками). Вот примеры общего конструктора и конструктора копирования для именованных и динамических переменных.

```
Cylinder c1(50,70);                    // вызывается общий конструктор
Cylinder c2=c1;                        // вызывается конструктор копирования
Cylinder *p = new Cylinder(50,70);     // вызывается общий конструктор
Cylinder *q = new Cylinder(*p);        // вызывается конструктор копирования
```

Для используемых по умолчанию конструкторов такой синтаксис недоступен. Применение круглых скобок при вызове в клиенте конструктора по умолчанию даст синтаксическую ошибку.

```
Cylinder c1();                         // синтаксическая ошибка
Cylinder c2;                           // вызывается конструктор по умолчанию
Cylinder *p = new Cylinder();          // синтаксическая ошибка: круглые скобки
Cylinder *q = new Cylinder;            // вызывается конструктор по умолчанию
```

Почему же такая несогласованность? Это чтобы легче было написать компилятор. Взгляните на первую строку последнего примера. Как узнать, что предполагается вызов конструктора, а не прототипа функции с именем c1() и возвращаемым типом Cylinder? Неизвестно. Разработчик компилятора тоже этого не знает. Один из способов избежать подобной неоднозначности состоит в запрете использования прототипа везде, кроме начала файла исходного кода. Вполне разумно, поскольку именно там обычно находятся прототипы. Между тем в C прототипы разрешается использовать повсеместно, а в C++ слишком ценится обратная совместимость, чтобы можно было позволить генерировать в этом случае синтаксическую ошибку. В Java не преследуется задача обратной совместимости с языком C, и синтаксис вызова конструктора по умолчанию в клиенте там согласован с синтаксисом вызова других конструкторов.

Конструкторы преобразования

Конструктор с одним параметром какого-то другого типа (не обязательно типа класса) называется конструктором преобразования. Часто он имеет тип одного из элементов данных класса. Конструктор преобразования полезен, когда клиент хочет задавать при создании каждого объекта значение только одного конкретного поля, а для других использовать значения по умолчанию.

Например, в программе моделирования может потребоваться создать объекты Cylinder с разными значениями радиуса. Первоначально — с нулевой высотой, которая потом растет, отражая процесс моделирования (рост артерий, связующих электронных компонентов, теплообмен через трубы отопления и т. д.).

```
Cylinder c1(50.0);                     // вызывается конструктор преобразования
Cylinder c2 = 30.0;                    // вызывается конструктор преобразования
```

Вновь, несмотря на разный синтаксис, оба оператора имеют один смысл — вызов конструктора преобразования.

В отличие от конструкторов по умолчанию и конструкторов копирования, конструкторы преобразования системой не подставляются. Если в классе определен конструктор преобразования с одним параметром типа double, то оба приведенных выше оператора дадут ошибку. Конструктор преобразования задает, что делать, если в качестве параметра указывается только одно значение, и какие значения использовать для других полей объекта. В следующем примере класс Cylinder определяет четыре конструктора: конструктор по умолчанию, конструктор копирования, конструктор преобразования и общий конструктор с двумя параметрами.

```
class Cylinder {
    double radius, height;
  public:
  Cylinder ()     // предусмотренный программистом конструктор по умолчанию
    { radius = 1.0, height = 0.0; }
  Cylinder (const Cylinder &c)              // конструктор копирования
    { radius = c.radius, height = c.height; }
  Cylinder (double r, double h)
    { radius = r, height = h; }             // общий конструктор
  Cylinder (double r)
    { radius = r, height = 0.0; }           // конструктор преобразования
  ....};
```

Конструктор преобразования — первый удар по системе строгого контроля типов в C++. Как уже упоминалось, все современные языки поддерживают строгий контроль типов. Если в каком-то контексте ожидается значение одного типа, то подстановка значения другого типа даст синтаксическую ошибку. Рассмотрим, к примеру, такой оператор:

```
Cylinder c2 = 30.0;                  // вызывается конструктор преобразования
```

Если Cylinder — это просто структура C, то такой оператор синтаксически ошибочен. Компилятор сообщает об этом и говорит, что у вас есть шанс подумать и решить, что вы хотите сделать. Если Cylinder — класс C++ без конструктора преобразования, тоже возникает синтаксическая ошибка. У вас также не будет возможности выполнить программу и проанализировать ее результаты. Когда Cylinder — класс C++ без конструктора преобразования, то синтаксической ошибки не будет. Если это сделано намеренно, то все замечательно. Если же нет, то компилятор не защитит от такой ошибки. Система строгого контроля типов здесь дает сбой.

В качестве следующего примера рассмотрим функцию CopyData() из этой главы (предполагая, что элементы данных radius и height объявлены как public).

```
void CopyData(Cylinder *to, const Cylinder &from)
                                          // копирование данных Cylinder
  { to->radius=from.radius; to->height=from.height; }  // запись со стрелкой
```

Для простой структуры C или для класса C++ без конструктора преобразования этот вызов функции в клиенте даст синтаксическую ошибку:

```
CopyData(&c2,70.0);                  // здесь пропущено FROM Cylinder
```

Если доступен конструктор преобразования, компилятор будет генерировать программный код, создающий временный неименованный объект Cylinder, вызывающий для этого временного объекта конструктор преобразования (с фактическим аргументом 70,0) и передающий временный неименованный объект функции CopyData() как второй аргумент.

Если в клиенте используется значение числового типа, отличного от double, это не проблема. Компилятор генерирует код, преобразующий данное числовое значение в double, а затем передающий это значение как фактический аргумент конструктору преобразования.

```
Cylinder c2 = 30;                    // 30 преобразуется в double
CopyData(&c2,70);                    // 70 преобразуется в double
```

Конечно, если это именно то, что требовалось написать, то можно отметить такое положительное свойство C++ как гибкие возможности реализации намерений программиста, но, если такой код написан по ошибке, остается лишь пожалеть, что компилятор не сообщил о ней, чтобы дать возможность исправить программу еще до ее выполнения.

Деструкторы

Объект C++ уничтожается в конце выполнения программы (для объектов static или extern), при достижении закрывающей фигурной скобки, завершающей область действия (для автоматических объектов), при выполнении операции delete (для динамических объектов с памятью, выделенной через new) или при вызове библиотечной функции free() (для объектов с памятью, выделенной через malloc()).

Когда уничтожается объект класса (за исключением вызова free()), непосредственно перед уничтожением вызывается деструктор класса. Если деструктор в классе не определен, то вызывается деструктор, подставляемый системой по умолчанию (как и конструктор по умолчанию, он ничего не делает).

Деструктор, определяемый программистом, аналогичен конструктору. Это функция-член класса. Синтаксис деструктора еще более строгий, чем синтаксис конструктора. Указывать возвращаемый функцией тип в ее интерфейсе недопустимо, а в теле функции не может присутствовать оператор return. Деструктор имеет то же имя, что и имя класса, но ему предшествует тильда (~), например ~Cylinder(). Деструкторы в отличие от конструкторов не могут иметь параметров.

Конструкторы и деструкторы — хороши для размещения операторов отладки.

```
class Cylinder {
  double radius, height;
public:
  ~Cylinder ()                          // определяемый программистом деструктор:
                                        // нет возвращаемого типа
  { cout << "Cylinder (" << radius << ", " << height
         << ") уничтожен" << endl; }        // нет возвращаемого значения
. . . . } ;
```

Когда деструктор реализуется вне области действия класса, используется операция области действия. Обратите внимание, что тильда — это часть имени функции, а не часть операции области действия.

```
Cylinder::~Cylinder ( )          // деструктор класса: нет возвращаемого типа
  { cout << "Cylinder (" << radius << ", " << height
         << ") уничтожен" << endl; }        // нет возвращаемого типа
```

Поскольку деструкторы не могут иметь параметров, перегрузка имен для них невозможна (т. к. функции сперегрузкой имен должны различаться по списку параметров). Следовательно, каждый класс может иметь не более одного деструктора.

Определяемый программистом конструктор нужен в том случае, когда объект использует динамическую память или другие ресурсы (файлы, блокировки базы данных и пр.). Чтобы избежать утечек ресурсов, деструктор должен возвращать эти ресурсы системе. Для таких последовательностей, как выделение и освобождение памяти, открытие и закрытие файлов и т. д., функции-деструкторы дополняют конструкторы.

Рассмотрим пример класса, где может быть полезен деструктор. Класс Name содержит строку символов — фамилию человека. Конструктор инициализирует символьный массив. (Это конструктор преобразования, так как он имеет один параметр с типом, отличным от Name.) Для простоты данные объявлены как public, и предусматривается только один метод show_name(), отображающий на экране содержимое объекта.

```
struct Name {
    char contents[30];      // фиксированный размер объекта, открытые данные
  Name (char* name);        // или Name(char name []);
  void show_name();
} ;                         // деструктор еще не нужен
```

```
Name::Name(char* name)              // конструктор преобразования
{ strcpy(contents, name); }         // стандартное действие: копирование
                                    // данных аргумента

void Name::show_name()
{ cout << contents << "\n"; }
```

Клиент может определять объекты данного типа и отображать их содержимое на экране.

```
Name n1("Джонс");                   // вызывается конструктор преобразования
Name *p = new Name("Смит");         // вызывается конструктор преобразования
n1.show_name(); p->show_name();
delete p;                           // удаляется неименованный объект
```

В этой конструкции, независимо от длины фамилии, выделяется один и тот же объем памяти. Когда фамилия короткая, память теряется напрасно, а если длинная — возможна порча содержимого памяти.

Популярным решением этой проблемы является динамическое распределение памяти. Вместо использования в качестве элементов данных фиксированного массива этот класс определяет только один указатель символьного типа. Объем выделяемой динамической памяти зависит от длины фамилии, передаваемой клиентом. В конструкторе для определения требуемого объема памяти вызывается функция strlen() (дополнительный символ — это завершающий ноль). Затем эта память выделяется и для инициализации динамически распределяемой области вызывается функция strcpy().

```
struct Name {
    char *contents;                 // указатель на динамически распределяемую
                                    // память: все равно public
  Name (char* name);                // или Name(char name []);
  void show_name();  } ;            // теперь деструктор нужен

Name::Name(char* name)              // конструктор преобразования
{ int len = strlen(name);           // аргумент - число символов
  contents = new char[len+1];       // выделение динамической памяти для аргумента
  if(contents == NULL)              // 'new' выполнена неуспешно
    { cout << "Нет памяти\n"; exit(1); }        // отказ
  strcpy(contents, name); }         // успех: копирование данных аргумента

void Name::show_name()
{ cout << contents << "\n"; }
```

Чтобы обсудить, что происходит при использовании новой версии класса Name, поместим программный код клиента в глобальную функцию.

```
void Client()
{ Name n1("Джонс");                 // вызывается конструктор преобразования
  Name *p = new Name("Смит");       // вызывается конструктор преобразования
  n1.show_name(); p->show_name();
  delete p;  }                      // удаляется неименованный объект
```

Когда в функции Client() выполняется оператор delete p;, освобождается память, на которую ссылается указатель p. Эта память содержит только указатель contents. Память по указателю contents не удаляется и становится недоступной. Это — утечка памяти. Обратите внимание, что оператор delete p; не удаляет указатель p. Он удаляет то, на что этот указатель указывает. Указатель p удаляется в соответствии с правилами области действия (когда завершается та область действия, где он определен). Это происходит, когда завершается выполнение функции Client() (достигается закрывающая фигурная скобка).

Аналогично когда завершается функция Client(), локальный объект n1 уничтожается и содержимое указателя contents возвращается в стек. Память, на которую ссылается указатель contents, не возвращается системе. Это — утечка памяти.

Использование конструкторов жизненно важно для классов, где управление ресурсами происходит динамически. Для поддержания целостности программы C++ нужны деструкторы. Деструктор вызывается при каждом уничтожении объекта по правилам области действия или по операции delete (но не с помощью вызова функции free()). Следовательно, деструктор — подходящее место для освобождения памяти (и других ресурсов), выделенной во время существования объекта (в основном это происходит в конструкторах, но иногда динамическое распределение памяти выполняется и в других функциях-членах).

```
Создан объект: Джонс
Создан объект: Смит
Джонс
Смит
Уничтожен объект: Смит
Уничтожен объект: Джонс
```

Рис. 9.3. *Вывод программы из листинга 9.3*

Деструктор для класса Name очень прост. В листинге 9.3 показан класс Name с деструктором, возвращающим динамическую память. Рис. 9.3 демонстрирует результат выполнения программы с выводом через отладочные операторы в конструкторе и деструкторе.

Листинг 9.3. Пример использования деструктора для возврата динамически распределяемой памяти, выделенной для именованных и неименованных объектов

```
#include <iostream>
using namespace std;

struct Name {
     char *contents;                          // указатель public на динамическую память
   Name (char* name);                         // или Name (char name[]);
   void show_name();
   ~Name(); } ;                               // деструктор устраняет утечку памяти

Name::Name(char* name)                        // конструктор преобразования
{ int len = strlen(name);                     // число символов
   contents = new char[len+1];                // выделение динамической памяти для аргумента
   if (contents == NULL)                      // выполнение 'new' неудачное
     { cout << "Нет памяти\n"; exit(1); }     // отказ
   strcpy(contents, name);                    // стандартные действия
   cout << "создан объект: " << contents << endl;          // отладка

void Name::show_name()
{ cout << contents << "\n"; }

Name::-Name()                                 // деструктор
{  cout << "уничтожен объект: " << contents << endl;       // отладка
   delete contents; }                         // освобождает динамическую память, а не
                                              // удаляет указатель contents

void Client()
{  Name n1("Джонс");                          // вызывается конструктор преобразования
   Name *p = new Name("Смит");                // вызывается конструктор преобразования
   n1.show_name(); p->show_name();
   delete p;                                  // удаляется неименованный объект
   }

int main ()
{  Client ();
   return 0;
   }
```

Когда функция Client() выполняет операцию delete p;, вызывается деструктор класса Name с оператором delete contents. При уничтожении функцией Client() объекта n1 вызывается деструктор, который также выполняет оператор delete contents. При этом устраняется утечка памяти.

Рис. 9.4 демонстрирует использование памяти функцией Client(). Рис. 9.4A показывает состояние памяти после создания указателей на именованный объект n1 и неименованный объект p. Числа показывают, что сначала для n1 выделяется пространство стека (по правилам области действия), затем выделяется динамическая память для "Джонс" (конструктором), память для неименованного объекта (на который указывает p) и, наконец, выделяется динамическая память для "Смит".

На рис. 9.4B и C представлена схема уничтожения объектов. Рис. 9.4B показывает, что сначала возвращается память, выделенная для "Смит" (с помощью деструктора), затем — память, выделенная для неименованного объекта (через операцию delete). Указатель p сохраняется, так как операция delete не удаляет указатель. Освобождается динамически распределяемая память, на которую ссылается указатель.

Рис. 9.4C демонстрирует правила освобождения памяти в стеке для указателя p и именованного объекта n1. "Кончина" указателя не влечет за собой никаких событий. Уничтожение объекта n1 приводит к вызову конструктора Name, освобождению динамически распределяемой памяти, выделенной для строки "Джонс" конструктором, и освобождению пространства стека, занимаемого n1.

Не пожалейте время, чтобы как следует разобраться в этом рисунке, и поэкспериментируйте со своим собственным программным кодом. Некоторые программисты считают, что динамически распределяемую память (в данном примере память, выделяемую для строк "Джонс" и "Смит") проще анализировать, если рассматривать ее как часть экземпляра объекта. Динамическая память — дополнительный ресурс, выделяемый каждому экземпляру объекта и позднее возвращаемый системе. С этой точки зрения память, выделяемая самому объекту, имеет размер, соответствующий размеру его элементов данных, а не аргументам конструктора. Но это дело вкуса.

Заметим, что если функции Client() не удается выполнить вызов delete p;, то объект, на который ссылается указатель p (его указатель contents и память по указателю contents), никогда не уничтожается и ресурсы не возвращаются

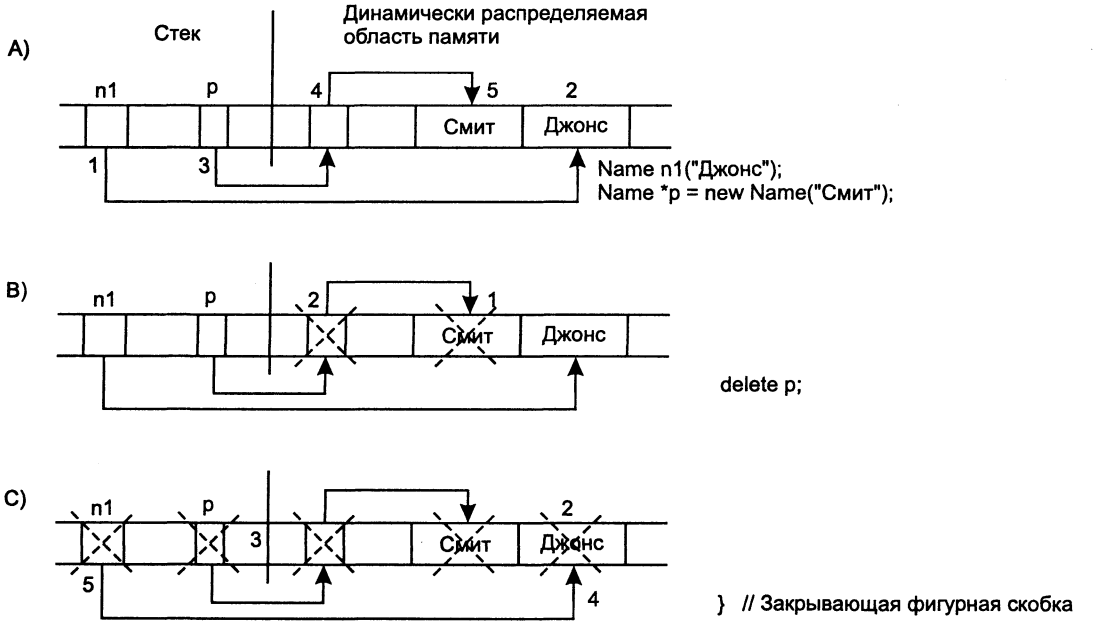


Рис. 9.4. *Схема управления памятью для функции-клиента Client() из листинга 9.3*

системе. Программист, отвечающий за клиента, должен обеспечить целостность программы. Для объектов, управляемых правилами области действия, такого требования нет. Например, объект n1 автоматически удаляется, когда при выполнении программы достигается закрывающая фигурная скобка функции Client(). Программисту не нужно прилагать никаких усилий, чтобы это произошло. Все это необходимо для управления памятью в том случае, если разработчик серверной части включает в класс конструктор Name.

Время вызова конструктора и деструктора

Термин "конструктор" предполагает, что эта функция создает объект. Термин "деструктор" предполагает, что данная функция объект уничтожает. Не попадайтесь в эту ловушку. Термины описывают конструкторы и деструкторы некорректно.

В предыдущем разделе было отмечено, что конструктор вызывается *после* создания объекта, а деструктор — *перед* его уничтожением. Часто в книгах по C++ не обращают внимание на подобное различие и утверждают, что конструкторы и деструкторы вызываются *при* создании и уничтожении объектов. И напрасно, поскольку программисты думают, что конструкторы создают объекты, а деструкторы — уничтожают их.

Это не так. За создание и уничтожение объектов отвечают правила области действия (для именованных объектов) или операции new и delete (для объектов неименованных). Конструкторы только инициализируют поля объекта после того, как они уже созданы, и распределены дополнительные ресурсы, например, динамическая память. Деструкторы лишь возвращают ресурсы, необходимые объектам во время их существования, например динамическую память, выделенную в конструкторах и других функциях.

Область действия класса и подмена имен во вложенных областях

Реальное время вызова конструктора/деструктора зависит от области действия и класса памяти экземпляров объектов.

Область действия определяет доступность переменных и объектов в разных частях программы. Класс памяти определяет срок жизни переменных и объектов от создания до уничтожения. В данном разделе мы продолжим обсуждение области действия и классов памяти, начатое в главе 6. Если вы почувствуете, что эта тема слишком сложна, пропустите ее при первом чтении (остается надеяться, что будет еще и второе чтение). Этот материал важен, но может подождать, пока вы наберетесь опыта написания и чтения исходных кодов C++.

Поскольку глобальные переменные можно определять в любом месте файла, даже после каких-либо определений функций, они недоступны в функциях, определяемых в файле раньше, до объявления переменной.

```
Cylinder Cglobal;          // доступна в любом месте файла

int main()
{ Cylinder c;              // область действия ограничена main()
  . . . . }

int y;                     // невидима в main(), видима в foo()

void foo()                 // не может вызываться из main(), если нет прототипа
{ y = 0;                   // доступ к глобальной переменной
  Cylinder Clobal;
  Clodal.setCylinder(10,30)                  // компоненты public видимы
  Cglobal.setCylinder(5,20); }               // компоненты public видимы
  . . . .                                    // Cglobal, y, foo() здесь видимы
```

В C++ все это законно, но не является хорошим стилем программирования.

Поскольку локальная переменная может определяться в любом месте блока (блока функции или неименованного блока), она недоступна в этом блоке до точки определения. Если имя локальной переменной совпадает с именем глобальной, то локальное имя используется в том блоке, где оно определяется, а глобальное — вне локального блока.

В дополнение к этим двум областям действия (о которых подробно рассказывалось в главе 6) C++ добавляет еще одну — область действия класса. Каждое имя, определенное в области действия класса (элементы данных, функции-члены, закрытый или общедоступный компонент класса), известно во всей области действия класса. Правила однопроходной компиляции, действующие для глобальных и локальных областей, к области действия класса не применяются, поэтому во всех примерах с классом Cylinder элементы данных доступны в функциях-членах этого класса независимо от порядка определения его компонентов.

Если определяемое в классе имя совпадает с глобальным именем, то все ссылки на него в области действия класса относятся к имени, определенному в области действия класса. Вне области действия класса (т. е. вне его функций-членов) ссылки на данное имя относятся к глобальному имени.

Когда имя, определяемое в области действия класса, совпадает с локальным именем, определенным в одной из его функций-членов, в данной функции применяются локальное имя, а имя, определенное в области действия всего класса, используется в других функциях-членах.

Короче, локальное имя может скрывать имя из области действия класса и глобальное имя, а имя, определенное в области действия класса — скрывать глобальное имя. Эти правила сокрытия имен могут переопределяться операцией глобальной области действия :: (для глобальных имен) и операцией области действия класса (для локальных имен).

В предлагаемом ниже примере имя radius используется для глобальной переменной, для элементов данных Cylinder и для локального имени в функции-члене setCylinder() класса Cylinder.

```
double radius = 100;                        // глобальное имя

struct Cylinder {                           // начало области действия класса
    double radius, heigth;                  // компонент radius скрывает
                                            // глобальное имя radius

void setCylinder(double r, double h)
{ double radius;
  radius = r; height = h;                   // локальное имя radius скрывает
                                            // элемент данных radius
  Cylinder::radius = radius; }              // область действия класса
                                            // переопределяет это правило

void scaleCylinder(double factor)
{ radius = ::radius;                        // глобальная операция области
                                            // действия переопределяет правило
  height *= factor; }
  . . . } ;                                 // конец области действия
```

Когда параметру r в setCylinder() присваивается значение radius, то происходит присваивание локальной переменной, а не элементу данных Cylinder. Чтобы присвоить значение элементу данных radius, нужно использовать операцию области действия класса. В функции setCylinder() radius означает элемент данных. Для получения значения глобальной переменной radius следует использовать операцию глобальной области действия.

Иногда программисты применяют для параметра метода и для элемента данных одно и то же имя. Например, в такой версии setCylinder() функция будет некорректна:

```
void Cylinder::setCylinder(double radius, double h)
                                                  // некорректная функция
{ radius = radius; height = h; }                  // локальный параметр
                                                  // скрывает элемент данных
```

Эта функция компилируется и выполняется без проблем, однако разработчик и компилятор понимают присваивание radius = radius; по-разному. Для разработчика radius слева означает элемент данных, а radius справа — параметр. Для компилятора radius с обеих сторон — параметр. Присваивать параметр самому себе не особенно полезно, но это один из примеров, когда компилятор отказывается думать за программиста и строить догадки. Хотите присваивать параметр самому себе? Пожалуйста, C++ не запрещает.

Класс памяти относится к сроку существования переменных (автоматических, внешних, статических): когда они создаются и уничтожаются.

Для локальных автоматических переменных выделяется память в стеке, когда при выполнении программы достигается их определение (размер памяти может зависеть от области действия). Если одно и то же имя используется в разных областях действия, то оно будет ссылаться на разные области памяти. При отсутствии инициализации содержимое памяти будет не определено. Для объекта конструктор вызывается сразу после выделения памяти.

Автоматические переменные уничтожаются, когда выполнение достигает конца блока, где они определены. Для объекта деструктор вызывается непосредственно перед возвращением памяти системе.

Память для внешних или статических (локальных или глобальных) переменных выделяется и инициализируется в фиксированной области перед началом выполнения программы. Если явное начальное значение после выделения памяти не указано, память инициализируется нулями. Для объекта перед началом выполнения main() вызывается конструктор (и все функции, которые он может вызывать). Порядок вызова конструкторов для разных объектов не определен.

Внешние и статические (локальные или глобальные) переменные уничтожаются, когда завершается выполнение функции main(), т. е. достигается закрывающая фигурная скобка, или программа прерывается другими путями. Для объектов деструктор вызывается непосредственно перед уничтожением объекта.

Хороший шанс продолжить обсуждение классов памяти, начатое в главе 6. Если программа не использует глобальных переменных или определяемых программистом классов, порядок выполнения ее кода будет четко известен. Выполнение начинается с первой строки main() и заканчивается ее последней строкой.

Память для динамических переменных выделяется и освобождается явно. Обычно операции new или delete либо функции malloc() и free() не вызываются в той же функции (области действия). Часто выделение памяти для динамической переменной происходит в одной функции, а ее освобождение — совсем в другой. (Очевидно, эти функции должны принадлежать одному классу клиента.)

Управление памятью с помощью операций и вызовов функций

В данном разделе сравнивается использование операций new и delete с применением функций malloc() и free(). Как и предыдущий раздел, вы можете пропустить его, если он покажется слишком технически сложным, но не забудьте вернуться к нему позднее и познакомиться с двумя аспектами:

1) рекомендацией по использованию операций new и delete в сравнении с malloc() и free();

2) критикой приведенного примера за несоблюдение принципов объектно-ориентированного программирования.

Отметим, что конструктор вызывается только после вызова объекта класса, созданного по правилам области действия или операцией new. Он не вызывается после malloc(). Аналогично деструктор вызывается только перед уничтожением объекта согласно правилам области действия или операции delete. Вызов функции free() не активизирует деструктор.

Если используются функции malloc() или free(), то программист, отвечающий за клиентскую часть, должен обеспечить возврат памяти в динамически распределяемую область, когда объекты станут не нужны. В клиенте нужно распределять динамическую память и позднее освобождать ее, возвращая в динамическую область. Нарушение этой обязанности приводит к порче содержимого памяти и ее "утечкам". Важно различать динамическое управление объектами и динамическое управление выделяемой для объектов памятью, когда элементы данных класса представляют собой указатель на динамическую память.

В листинге 9.4 показан пример, аналогичный приведенному в листинге 9.3, но вместо new для выделения памяти объекту используется функция malloc(). Она выделяется динамически через указатель p. Очевидно, управление памятью здесь будет более сложным. Клиент распределяет в динамической области память для неименованного объекта и затем использует ее для содержимого объекта (строка "Смит"). Результат выполнения данного примера будет тем же, что и программы из листинга 9.3. (Здесь выключены операторы отладки в конструкторе и деструкторе.)

Листинг 9.4. Управление памятью в клиенте, а не в сервере

```
#include <iostream>
using namespace std;

struct Name {
     char *contents;                          // указатель public на динамическую память
   Name (char* name);                         // или Name (char name[]);
   void show_name();
    ~Name(); } ;                              // деструктор исключает утечки памяти

Name::Name(char* name)                        // конструктор преобразования
{  int len = strlen(name);                    // число символов
   contents = new char[len+1];               // выделение динамической памяти для аргумента
   if(contents == NULL)                       // неудачное выполнение 'new'
     { cout << "Нет памяти\n"; exit(1); }     // отказ
   strcpy(contents, name); }                  // стандартные действия

void Name::show_name()
{ cout << contents << "\n"; }

Name::-Name()                                 // деструктор освобождает динамическую
{ delete contents; }                          // память, а не удаляет указатель contents

void Client()
{  Name n1("Джонс");                          // вызывается конструктор преобразования
   Name *p = (Name)malloc(sizeof(Name));      // не вызывается конструктор преобразования
   p->contents = new char[strlen("Смит")+1];  // распределение памяти
   if (p->contents == NULL)                   // неудачное выполнение 'new'
      { cout << "Нет памяти\n"; exit(1); }    // отказ
   strcpy(p->contents, "Смит");               // стандартные действия
   n1.show_name(); p->show_name();            // использование объектов
   delete p->contents;                        // для избежания утечек памяти
```

```
    free (p);                          // обратите внимание на последовательность действий
}                                      // p удаляется, вызывается деструктор для объекта n1

int main()                             // перенос обязанностей на функции-серверы
{ Client();
  return 0;
}
```

В данном примере объект n1 создается согласно правилам области действия, конструктор соответственно инициализирует для него динамическую память. Неименованный объект, на который ссылается указатель p, распределяется с помощью функции malloc(), и конструктор не вызывается. При вызове функции malloc() выделяется только память для объекта (указатель p->contents). Дополнительная память из динамически распределяемой области, необходимая для хранения информации (фамилии), не выделяется. Следовательно, клиент распределяет и инициализирует память в динамической распределяемой области, на которую ссылается p->contents.

Когда функция Client() завершает работу, об удалении объекта n1 и возврате динамически распределяемой памяти можно не беспокоиться. Он уничтожается по правилам области действия, и память возвращается деструктором. Для неименованного объекта, на который указывает p, все иначе. В клиенте его не только нужно уничтожать, но и возвращать используемую объектом динамически распределяемую память.

В этом небольшом примере применяются классы, объекты, сообщения, динамическое управление памятью, конструкторы и деструкторы — весь впечатляющий арсенал программирования на C++. Одновременно здесь нарушаются все принципы объектно-ориентированного программирования. Единственное оправдание в том, что это сделано намеренно. Однако часто программисты делают это, сами того не замечая. Давайте снова разберем весь перечень грехов.

В примере нарушен принцип инкапсуляции: клиент использует имена полей объекта contents, создавая тем самым дополнительную зависимость. Если имя поля класса Name изменится, то придется изменять и функцию Client().

Нарушен также принцип сокрытия информации (как это обсуждалось в главе 8). Клиент знает, что класс с именем Name использует динамически распределяемую память, а не символьный массив фиксированного размера. Если изменится архитектура класса Name, это повлияет и на функцию Client().

Такие зависимости вынуждают координировать действия разработчиков и программистов. У разработчиков придется выяснить все детали по классу Name, такие, как имена полей, управление динамической памятью и еще бог знает что. И это вместо простого изучения интерфейса функций, которым можно обойтись при использовании переменной n1.

Код функции Client() не выражается в терминах вызовов функций-членов класса Name. Вместо этого он содержит многочисленные операции доступа к данным и манипуляции с данными, поэтому при сопровождении программы придется потратить дополнительное время, пытаясь уяснить, какие именно цели в ней преследуются.

Хуже всего то, что обязанности не переносятся на класс сервера, хотя весь необходимый для этого сервис имеется. Распределение и освобождение памяти осуществляется в клиенте, а не с помощью объекта сервера.

Результат весьма плачевный. Клиент получился намного более сложным, чем следовало. Кроме того, здесь легко сделать ошибку — достаточно внести небольшие изменения в функцию Client(). В данной версии клиента сначала освобождается память, на которую указывает объект p, а затем делается попытка освободить динамически распределяемую память. При выполнении программы операционная система обвиняет ее в нарушении доступа к памяти и принудительно завершает.

Это разумно, так как объект, указываемый p, исчез, а значит, исчез и указатель p->contents. Не каждая ОС может позволить себе роскошь проверки всех обращений к памяти за счет скорости работы, и на многих платформах такая ошибка останется незамеченной.

```
void Client()
{ Name n1("Джонс");                  // вызывается конструктор преобразования
  Name *p=(Name*)malloc(unsigned(sizeof(Name)));
                                     // конструктор преобразования не вызывается
  p->contents = new char[strlen("Смит")+1];
                                     // распределение динамической памяти
  if (p->contents == NULL)                    // вызов 'new' был неуспешным
    { cout << "Нет памяти\n"; exit(1); }      // отказ
  strcpy(p->contents, "Смит");                // вызов 'new' был успешным
  n1.show_name(); p->show_name();             // использование объектов
  free (p);                                // неверная последовательность действий!
  delete p->contents;                         // удалять здесь нечего!
}                                       // p удален, вызван деструктор для объекта n1
```

Кроме того, если память для объекта распределяется с помощью new, то ее следует освобождать по delete. Использование функции free() — семантическая ошибка. Семантической ошибкой является и применение операции delete для возврата памяти, выделенной по функции malloc().

Обратите внимание, что семантическая ошибка отлична от синтаксической. Она может выражаться в принудительном завершении программы при ее выполнении или в некорректных результатах (выявляемых с помощью тестирования). Принцип "семантически некорректной программы" — неудачный вклад C++ в программную инженерию. Результаты некорректной последовательности вызовов "не определены", и программисту придется самому обеспечить отсутствие в программе бомбы замедленного действия, способной взорваться в любой момент.

Две характеристики функций malloc() и free() — отсутствие вызовов конструкторов/деструкторов и опасность получения некорректной программы при совмещении с операциями new и delete — могут привести к проблемам, поэтому применение функций malloc() и free() для динамического распределения памяти в C++ непопулярно. Однако оно очень популярно в языке C (где нет операций new и delete). Эти функции часто используются в унаследованных системах, а также в приложениях, динамически обрабатывающих множество операций с памятью для повышения производительности. Функции malloc() и free() можно применять для создания в избранных классах специализированных операций new и delete. О таком "продвинутом" использовании операций рассказывается далее.

Версия, представленная в листинге 9.3, лучше, чем версия из листинга 9.4: она не жертвует инкапсуляцией, не нарушает принцип сокрытия информации и не создает необходимости дополнительной координации работы программистов. Между тем она обременяет клиента обязанностями выделения и освобождения памяти для объекта Name, на который ссылается указатель p. Программисты часто используют динамическое управление памятью, когда оно не особенно полезно. Это как раз тот самый случай. Память для данного объекта можно выделять и освобождать с помощью правил области действия, а не явного управления.

```
void Client()
{ Name n1("Джонс");          // вызывается конструктор преобразования
  Name n2("Смит");           // нет динамического выделения/освобождения памяти
  n1.show_name(); n2.show_name();
  }                          // вызывается деструктор для объектов n1 и n2
```

Убедитесь, что ваши программы на C++ оказались не сложнее, чем могли бы быть.

Использование в коде клиента возвращаемых объектов

Функции C++ могут возвращать встроенные значения, указатели, ссылки и объекты, но не могут возвращать массивы (возвращаемые массивы можно имитировать возвратом указателей). Встроенные значения разрешается использовать только как r-значения. Другие возвращаемые значения (указатели, ссылки и объекты) допускается использовать как l-значения. Это открывает интересные возможности, делает исходный код C++ более выразительным, но в чем-то усложняет его понимание.

Материал данной главы можно легко пропустить при первом чтении, хотя обсуждаемые здесь принципы программирования весьма распространены.

Возврат указателей и ссылок

Начнем обсуждение с простых (несоставных) встроенных типов. Значения этих элементарных типов применяются как r-значения, а указатели и ссылки можно использовать как r- и l-значения.

Рассмотрим следующую версию класса `Point`. Его функция `setPoint()` изменяет состояние целевого объекта `Point`, а функции `getX()` и `getY()` возвращают целочисленное значение. Функция `getPtr()` возвращает указатель на элемент данных `x`, а `getRef()` — ссылку на элемент данных `x`. Здесь не предусмотрено функций, возвращающих указатель и ссылку на элемент данных `y`, так как функций `getPtr()` и `getRef()` достаточно для иллюстрации разных вопросов, включая модификацию состояния объекта.

```
class Point
{ int x, y;                          // закрытые данные
  public:
    void setPoint(int a, int b)
    { x = a; y = b; }
  int getX()                         // возвращает значение
    { return x; }
  int getY()
    { return y; }
  int* getPtr()                      // возвращает указатель на значение
    { return &x; }
  int& getRef()                      // возвращает ссылку на значение
    { return x; } } ;                // нет операции получения адреса для ссылки
```

Чтобы решить, следует ли использовать операцию получения адреса, нужно применять ту же логику, что и при присваивании или передаче параметра. Функция `getPtr()` возвращает указатель. Следовательно, при возврате значения `x` было бы несоответствие типов — синтаксическая ошибка. Функция `getRef()` возвращает ссылку, и эта ссылка может (и должна) инициализироваться значением, на которое она будет указывать до конца своей жизни. Следовательно, использование `&x` дало бы несоответствие типов (синтаксическая ошибка). Это адрес, а не значение `int`.

Когда функция возвращает значение, ее можно использовать только как r-значение в правой части присваивания или сравнения (или как входной параметр в вызове функции). В данном примере клиент манипулирует с возвращаемым функцией значением. Значение изменяется, но объект, чье значение возвращает функция, не модифицируется, поскольку при возврате по значению создается копия значения-оригинала (подобно передаче параметров по значению):

```
Point pt; pt.setPoint(20,40);
int a = pt.getX(), b = 2* pt.getY() + 4;      // OK, используется как r-значение
a += 10;                                      // 'a' изменяется, но pt.x нет
```

Возвращаемые функцией указатель или ссылку можно использовать как r-значение и как l-значение в левой части присваивания либо как входной параметр при вызове функции. В следующем примере первая строка интерпретирует возвращаемые функциями `getPtr()` и `getRef()` результаты как r-значения. Ничего необычного. Вторая строка модифицирует значения, на которые ссылаются указатель `ptr` и ссылка `ref`. Обратите внимание, что оба они указывают на данные переменной `pt` (т. е. `pt.x`). Эти данные — закрытые, но клиент может изменять их без помощи функций доступа. Третья строка использует вызов функции как l-значение и изменяет состояние переменной `pt`. Скобки в выражении `*pt.getPtr();` не нужны. Операция-селектор имеет более высокий приоритет, чем операция разыменования, применяемая здесь для разыменования возвращаемого методом значения, а не указателя на целевой объект (`pt` — не указатель, это имя объекта `Point`).

```
int *ptr = pt.getPtr(); int &ref = pt.getRef();
                                              // OK, используется как r-значение
*ptr += 10; ref += 10;          // закрытые данные изменяются через псевдонимы
*pt.getPtr()=50; pt.getRef()=100;                  // закрытые данные изменены
```

Во-первых, данный синтаксис использования функции как l-значения необычен. Во-вторых, такая практика, как некоторые могут сказать, есть нарушение инкапсуляции и сокрытия информации: изменяются закрытые данные, которые не должны быть доступны в клиенте. Но кто сказал, что под сокрытием информации подразумевается неизменение закрытых данных? Вызов функции `setPoint()` не изменяет закрытые данные и не нарушает сокрытия информации, как и вызов `getRef()`. Инкапсуляция и сокрытие информации — это вопрос предотвращения зависимости между классами, а не предотвращение изменения закрытых данных.

С точки зрения разработки ПО основная проблема этого примера в том, что здесь используются псевдонимы. Псевдонимы также ссылаются на элемент данных `x`, но используют для этого другие имена: `ptr`, `ref`, `getPtr()` и `getRef()`. Между тем эти имена, особенно `getPtr()` и `getRef()`, никак не показывают, что они ссылаются на `x`. Следовательно, такой подход вынуждает тратить дополнительное время на понимание программы при ее сопровождении. Используйте его осторожно, если это вообще стоит делать. В C++ данный метод допустим, но опасен. Он может принести еще больше вреда, чем применение глобальных переменных.

Возврат указателей и ссылок требует, чтобы передаваемый в вызывающую программу адрес после завершения функции был допустимым. В предыдущем примере функции `getPtr()` и `getRef()` возвращают указатель на `pt.x`, а `pt.x` остается в области действия после завершения функции. Иногда это не так. В следующем примере функции `getDistPtr()` и `getDistRef()` вычисляют расстояние между целевым объектом `Point` и началом координат. А возвращают указатель и ссылку на вычисленное значение расстояния. Какой досадный просчет!

```
class Point
{ int x, y;
public:
    . . .                    // setPoint(), getX(), getY(), getPtr(), getRef()
    int* getDistPtr()
    {  int dist = (int)sqrt(x*x + y*y);
       return &dist; }                 // нет копирования, но dist исчезла
    int& getDistRef()
    {  int dist = (int)sqrt(x*x + y*y);
       return dist; } } ;              // другой синтаксис, некоторые проблемы
```

Локальная переменная dist исчезает после завершения функций getDistPtr() и getDistRef(). Использование ее адреса может дать корректный результат, если занимаемая переменной память не применяется для чего-то еще, в противном случае результаты вычислений окажутся некорректными. Некоторые компиляторы могут при этом давать предупреждение, другие — нет. Как бы то ни было, данная версия класса Point (приведенная выше) и программы-клиента (ниже) синтаксически корректны.

```
Point pt; pt.setPoint(20,40);
int * ptr = pt.getDistPtr();                          // недопустимый указатель
cout << " Указатель на расстояние : " << *ptr << endl;      // OK
int &ref = pt.getDistRef();                          // недопустимая ссылка
cout << " Ссылка на расстояние: " << ref << endl;           // OK
cout << " Указатель на расстояние : " << *ptr << endl;      // плохо
cout << " Ссылка на расстояние: " << ref << endl;           // плохо
```

```
Указатель на расстояние: 44
Ссылка на расстояние: 44
Указатель на расстояние: 4198928
Ссылка на расстояние: 4198928
```

Рис. 9.5. *Корректные и некорректные результаты возврата указателя и ссылки*

Результаты выполнения данного примера на моей машине представлены на рис. 9.5. С первым случаем использования недопустимого указателя и ссылки все обошлось: оба значения корректны, хотя ни ссылка, ни указатель допустимыми не являются. Попытка снова вывести значения дает некорректные результаты. Это говорит о том, что любое другое их использование некорректно. Такие ошибки могут оставаться незамеченными. Разве можно, протестировав значения ref и *ptr и увидев корректные результаты, ожидать, что они могут измениться? Бдительность программиста, очевидно, будет направлена на другие вопросы. А ваша?

Все мы воспринимаем корректные результаты выполнения программы как свидетельство ее правильности. Конечно, может возникнуть желание протестировать программу на другом наборе данных и охватить дополнительные пути ее алгоритма, но повторять тесты для одних и тех же входных данных непродуктивно, да и в голову никому не придет. Ведь результаты должны быть такими же. Но это в других языках программирования. Конечно, и в C++ тоже, но только если вы знаете, что делаете.



Осторожно! Убедитесь, что при возврате указателя или ссылки из функции они не указывают на память, уже ставшую недействительной по правилам области действия C++. Нарушение данной рекомендации не приведет к синтаксической ошибке и может не проявиться в виде некорректных результатов. Не всегда следует интерпретировать правильные результаты как свидетельство корректности программы.

В общем случае полезно ограничиться использованием в качестве возвращаемых значений булевых флагов, которые говорят об успешном или неуспешном выполнении той или иной функции. Однако эстетическая привлекательность таких функций, как getX() и getY(), велика, и программисты всегда будут с ними работать. Не увлекайтесь мощными возможностями C++ и не возвращайте указателей или ссылок, особенно на значения, которые могут стать недействительными. Компилятор не защитит вас от такой ошибки.

Возврат объектов

В предлагаемом далее примере к классу Point добавлены еще три функции: closestPointVal(), closestPointPtr() и closestPointRef(). Каждая такая функция воспринимает в качестве параметра ссылку на объект Point и вычисляет для параметра и для адресата расстояние до начала координат. Если объект-параметр

находится ближе к началу координат, то функция возвращает в этот объект, а если к начальной точке ближе адресат сообщения, то функция возвращает целевой объект (как разыменованный указатель this, ссылающийся на него).

Первая функция возвращает сам объект, вторая — указатель на ближайший объект, а третья — ссылку на ближайший объект. Преимущество данного интерфейса в том, что он позволяет создавать цепочку сообщений, когда значение, возвращаемое одной функцией, используется как цель для вызова другой. Все три вида возвращаемых значений (объекты, указатели и ссылки) можно использовать как r- и l-значения.

```
class Point
{ int x, y;
public:
  . . .                        // setPoint(), getX(), getY(), getPtr(), getRef()
  . . .                        // getDistPtr(), getDistRef()
  Point closestPointVal(Point& pt)
  { if (x*x + y*y < pt.x *pt.x + pt.y * pt.y)
       return *this;           // значение объекта: копирование в объект temp
    else
       return pt; }            // значение объекта: копирование в объект temp
  Point* closestPointPtr(Point& p)
                               // возвращает указатель: нет копирования
  { return (x*x + y*y < p.x*p.x + p.y* p.y) ? this : &p; }
  Point& closestPointRef(Point& p)
                               // возвращает ссылку: нет копирования
  { return (x*x + y*y < p.x*p.x + p.y* p.y) ? *this : p; } } ;
```

Здесь this — ключевое слово, обозначающее указатель на целевой объект сообщения. В следующем примере это объект p1. Первая функция использует обыкновенную запись (два оператора return), а последние две — сокращенную (условный оператор).

Обратите внимание, как режимы адресации отражаются на возвращаемых объектах. Функция closestPointVal() возвращает объект Point (по значению). Когда возвращается целевой объект, указатель this (ссылающийся на цель) нужно разыменовать, а поля целевого объекта p1 — скопировать в поля принимающего (объекта pt). Эта ссылка — синоним объекта, на который она указывает (объект p2), а поля данного объекта копируются в принимающий объект (объект pt).

```
Point p1,p2; p1.setPoint(20,40); p2.setPoint(30,50);
                                        // присваивание для объектов Point
Point pt = p1.closestPointVal(p2);      // копируются поля ближайшего объекта
```

Функция closestPointPtr() возвращает указатель на ближайший объект Point. Если целевой объект ближе, чем объект, заданный параметром, возвращается указатель this (ссылающийся на цель, например p1). В следующем примере значение указателя копируется в принимающий указатель (p). Если ближе объект, заданный параметром, то используется его ссылка p. Так как данная ссылка — синоним указываемого ею объекта (а не адреса этого объекта), в принимающий указатель копируется значение &p. Этот указатель можно использовать для доступа к компонентам ближайшего объекта (p1 или p2).

```
Point *p = p1.closestPointPtr(p2);
                                   // возвращается указатель: быстрый способ
p->setPoint(0,0);                  // перемещение p1 или p2 в начало координат
```

Как можно видеть, возврат значения-объекта — потенциально медленный метод, и возврат указателя на объект позволяет избежать копирования полей объекта. Когда возвращается ссылка на объект, ситуация недостаточно ясна. Если ближайшим является целевой объект, функция `closestPointRef()` возвращает ссылку на ближайший объект `Point`. Когда ближе находится целевой объект, нужно использовать указатель `this`. Так как нельзя присваивать указатель ссылке, следует применять для целевого объекта запись `*this`, однако нужно иметь в виду, что это не означает, что создается копия целевого объекта. Это только запись. Подобно передаче параметров по ссылке, копируется лишь адрес (ссылка), а не поля объекта. Если ближе объект, заданный параметром, то его ссылка `p` используется непосредственно и с тем же результатом — копируется лишь ссылка, но не поля.

```
Point &r = p1.closestPointRef(p2);      // возвращается ссылка: быстрый метод
r.setPoint(0,0);                  // перемещение p1 или p2 в начало координат
```

Если же принимающая переменная в клиенте представляет один из объектных типов, а не ссылочный тип, то имеет место копирование:

```
Point pt = p1.closestPointRef(p2);          // p1 или p2 копируется в pt
```

Как можно видеть, при возврате ссылки на объект не всегда решаются проблемы производительности.

Основной стимул возврата ссылки на объект (когда результат возвращается по значению, по ссылке или по указателю) — возможность организации цепочки сообщений, передачи сообщения возвращаемому функцией объекту.

```
Point p1, p2; p1.setPoint(20,40); p2.setPoint(30,50);
int a = p1.closestPointVal(p2).getX();              // более медленный способ
int b = (*p1.closestPointPtr(p2)).getX();           // быстро и элегантно
int c = p1.closestPointRef(p2)).getX();             // быстро и элегантно
```

Здесь объект, возвращаемый функцией `closestPointVal()`, представляет собой временный неименованный объект `Point`. Он существует достаточно долго, чтобы передать сообщение функции `getX()`, после чего исчезает. В двух других вызовах функции указатель и ссылка обозначают один из объектов, определенных в клиенте. Вопрос срока жизни такого объекта не стоит — он существует здесь и все.

В предыдущем примере сообщение, переданное возвращаемому объекту, не изменяло состояние этого объекта. Цепочечную запись можно использовать также для передачи сообщений, меняющих состояние целевого объекта:

```
p1.closestPointRef(p2).setPoint(15,35);
                                    // что здесь устанавливается? p1? p2?
p1.closestPointPtr(p2)->setPoint(10,30);      // что здесь устанавливается?
```

В первом из этих примеров клиент изменяет объект `p1` или `p2` и изменения сохраняются. Во второй строке изменяется временный неименованный объект, который немедленно уничтожается! Такая операция совершенно бесполезна, но в C++ вполне законна.

```
p1.closestPointVal(p2).setPoint(0,0);      // создание объекта, присваивание
                                  // значений полям и уничтожение объекта
```

Нужно очень внимательно возвращать объекты. Часто выигрыш в производительности или удобство записи не стоят того, чтобы подвергать риску целостность программы и путаться потом в результатах операции. Кроме того, создание и уничтожение неименованных объектов требует времени, которое уходит как на управление динамически распределяемой областью, так и на вызовы конструкторов/деструкторов.

Еще о ключевом слове const

Данный раздел очень важен. В нем анализируется разный смысл ключевого слова const и показывается, как использовать его для решений одной из наиболее важных задач создания ПО — передачи идей разработчика о свойствах программных компонентов, чтобы их можно было использовать при сопровождении программы. Неудачное ее решение — один из простейших (и самых распространенных) способов внести вклад в кризис ПО.

Как указывалось ранее (в главах 4 и 7), ключевое слово const может иметь в языке C++ разный смысл. Он зависит от контекста. Когда это ключевое слово предшествует имени типа переменной, оно указывает, что значение переменной остается постоянным. Переменная должна инициализироваться в определении, и любая попытка присвоить ей другое (или даже то же самое) значение будет помечена как синтаксическая ошибка.

```
const int = 5;     // x не будет (и не может) изменяться
x = 20;            // синтаксическая ошибка: нельзя изменять x
int *y = &x;       // синтаксическая ошибка: предотвращает будущие изменения x
```

Когда указатель ссылается на "постоянную переменную" (невольный каламбур — она называется переменной, хотя на самом деле не изменяется), его нужно пометить как указатель на константу, для чего используется ключевое слово const, предшествующее имени типа. После этого любые попытки разыменования указателя как l-значения будут давать синтаксическую ошибку.

```
const int *p1 = &x;  // OK, *p1 не будет использоваться для изменения x
*p1 = 0;             // синтаксическая ошибка: *p1 не может быть l-значением
int a = 5;           // обычная переменная: ее можно изменять
p1 = &a; *p1 = 0;    // синтаксическая ошибка: 'a' нельзя изменять через *p1
```

Когда ссылка указывает на переменную const, ее нужно пометить как ссылку на константу. Для этого перед именем типа указывается ключевое слово const, после чего любая попытка использовать ссылку как l-значение считается синтаксической ошибкой.

```
int &r1 = x;          // синтаксическая ошибка: x не должна изменяться через r1
const int &r2 = x;    // OK, ссылка на константу, x изменяться не будет
r2 = 0;               // синтаксическая ошибка: r2 - ссылка на константу
const int &r3 = a;    // 'a' может изменяться, но не через r3
r3 = 0;             // синтаксическая ошибка: 'a' не может изменяться через r3
```

Когда ключевое слово const следует за операцией указателя, это означает, что указатель имеет тип const. Такой указатель должен все время ссылаться на одно и то же место, его нельзя переназначить на другой адрес памяти. Однако нет никакой гарантии, что не будет изменяться само значение по указателю.

```
int* const p2 = &a;        // p2 будет указывать только на 'a'
*p2 = 0;                   // OK: нет никакой гарантии, что это будет const
int b = 5; p2 = &b;        // синтаксическая ошибка: нарушение обязательств
```

В специальной записи, показывающей, что ссылка представляет константу, нет необходимости. В C++ ссылки являются константами по умолчанию. Их нельзя переустановить на другой адрес памяти. Как и в случае указателей, само расположенное по ссылке значение может изменяться:

```
int& r4 = a;          // r4 указывает только на 'a', const здесь не нужно
r4 = b;               // нет синтаксической ошибки, r не переназначено
```

Применение ключевого слова const в интерфейсах функций аналогично его использованию со значениями и указателями. Оно утверждает, что данный фактический аргумент (или указатель) в результате вызова не изменяется.

```
void f1(const int& x);      // x не изменяется функцией
void f2(const int x);       // лишнее: x в любом случае передается по значению
void f3(int* const y);      // лишнее: y передается по значению
void f4(int * const *y);    // OK, указатель передается по указателю
void f5(const int *&y);     // OK, указатель передается по ссылке
```

Применение объектов const в интерфейсах функции может затруднить для функции возврат указателя или ссылки на переданный в параметре объект. Если разрешить это, то клиент мог бы изменять данный объект через псевдонимы (подобно примерам, уже обсуждавшимся в этой главе выше).

Если вы пропустили данный раздел, то позвольте напомнить, что то же самое делают три функции: closestPointVal(), closestPointPtr() и closestPointRef(). Каждая из них сравнивает расстояние между целевым объектом и началом координат, между объектом, переданным в параметре, и начальной точкой. Если целевой объект ближе к началу координат, то каждая функция возвращает целевой объект. Если ближе объект-параметр, то каждая функция возвращает объект-параметр. Разница в том, что closestPointVal() возвращает сам объект, closestPointPtr() возвращает указатель на объект, а closestPointRef() — ссылку на объект. В разделе, посвященном возврату объектов в клиенте, для параметра функции не использовалось ключевое слово const. В следующем примере оно добавлено.

```
class Point
{ int x, y;                                  // закрытые данные
public:                                      // общие операции
  . . .                                      // setPoint(), getX(), getY(),
                                             // getPtr(), getRef()
  . . .                                      // getDistPtr(), getDistRef()
  Point closestPointVal(const Point& pt)     // не подходит: данные копируются
  { if (x*x + y*y < pt.x *pt.x + pt.y * pt.y)
       return *this;                         // значение объекта:
                                             // копирование в объект temp
    else
       return pt; }                          // значение объекта:
                                             // копирование в объект temp
  Point closestPointPtr (const Point& pt)                  // параметр const
  { return (x*x + y*y < p.x*p.x + p.y* p.y) ? this : &p; }   // ошибка
  Point& closestPointRef (const Point& p)                  // параметр const
  { return (x*x + y*y < p.x*p.x + p.y* p.y) ? *this : p; } } ;    // ошибка
```

Функция closestPointVal() возвращает либо целевой объект, либо объект-параметр, но в любом случае это копия объекта Point. Следовательно, ключевое слово const для параметра функции не ограничивает ее использование. Если клиент изменяет возвращенный объект, это изменение копии фактического аргумента, а не объекта, который должен оставаться неизмененным.

```
Point p1,p2;
p1.setPoint(20,40); p2.setPoint(30,50);
                                    // устанавливает значения полей Point
Point pt = p1.closestPointVal(p2);
pt.setPoint(0,0);                   // нет нарушения обязательств
p1.closestPointVal(p2).setPoint(0,0);   // не вредит, но и пользы никакой
```

Функция closestPointPtr() способна возвращать указатель на свой аргумент Point. Этот указатель может затем использоваться клиентом для изменения состояния объекта-аргумента. Аналогично функция closestPointRef() может возвращать ссылку на свой аргумент Point. Данную ссылку разрешается использовать для изменения состояния объекта-аргумента.

```
Point *p = p1.closestPointPtr(p2);                    // p2 не должен изменяться
p->setPoint(0,0);            // p2 может измениться - нарушение обязательств
Point &r = p1.closestPointRef(p2);                    // p2 не должен изменяться
r.setPoint(10,10);           // p2 может измениться - нарушение обязательств
```

В данном примере объект p2 реально не изменяется, так как все три функции возвращают объект p1, который ближе к точке начала координат, чем p2. Даже если объект p2 модифицируется, то это происходит вне функций closestPointRef() и closestPointPtr()! Тем не менее C++ не допускает такое использование объектов-констант. Компилятору очень трудно выявить подобные нарушения при анализе кода клиента (да и человеку — тоже), поэтому он объявляет обе функции ошибочными.

Формальная причина состоит в том, что объект-параметр (например, в функции closestPointPtr()) имеет ключевое слово const, а возвращаемый тип — нет.

```
Point* closestPointPtr(const Point& p)    // несогласованность: вредит const
{ return (x*x + y*y < p.x*p.x + p.y* p.y) ? this : &p; }
                                          //синтаксическая ошибка
```

C++ предлагает три способа выхода из данной ситуации. Один из них — исключить ключевое слово const из интерфейса функции. Второй способ избежать синтаксической ошибки состоит в применении операции const_cast, подавляющей свойство const в функциях-членах. Третий способ — использовать ключевое слово const двумя другими способами.

Исключение ключевого слова const из интерфейса функции — это для малодушных. Настоящий разработчик никогда не откажется от возможности сообщить программисту, занимающемуся кодом клиента или сопровождением программы, что он имел в виду при создании класса. Данная функция не изменяет свой объект-параметр, следовательно, здесь должно быть ключевое слово const.

Второй метод сложнее. Операция const_cast преобразует свой аргумент-константу в тот же тип, снимая защиту от изменений. Тип задается в угловых скобках между операцией const_cast и аргументом. Например выражение, записанное таким образом: const_cast<тип_значения>(значение-константа), приводит аргумент "значение-константа" типа "тип_значения" к тому же типу "тип_значения", но снимает защиту от изменений. Для класса Point с помощью выражения const_cast<Point*>(&p) указатель на постоянный объект Point преобразуется в указатель на непостоянный объект Point.

Вот версия класса Point, в которой отменяется свойство const аргумента при возврате значения из функций-членов:

```
class Point
{ int x,y;
public:
  . . .                    //,setPoint(), getX(), getY(), getPtr(), getRef()
  . . .                     // getDistPtr(), getDistRef() closestPointVal()
  Point* closestPointPtr(const Point& p)        // предотвращает изменение p
  { return (x*x + y*y < p.x*p.x + p.y *p.y) ? this : const_cast<Point*>(&p); }
  Point* closestPointRef(const Point& p)        // предотвращает изменение p
{ return (x*x + y*y < p.x*p.x + p.y*p.y) ? *this : const_cast<Point&>(p);
} } ;
```

Теперь в клиенте допускается изменение и возврат объектов, но это решение методом "грубой силы". Использовать его не стоит, хотя это лучше, чем удаление ключевого слова const из параметра функции, так как без const параметр в функции можно использовать как угодно. Применение ключевого слова const_cast снимает защиту только для данной конкретной операции (в нашем примере — возврат значения), но не в общем случае. Однако это довольно неуклюжий и не очень понятный метод.

Лучший способ для успешной компиляции closestPointPtr() и closestPointRef() — не изменять возвращаемый объект, объявив его константой. Для этого нужно поставить перед возвращаемым функцией значением ключевое слово const. Это и есть третий смысл ключевого слова const (о четвертом — чуть далее). При использовании с возвращаемым функцией значением оно предотвращает модификацию этого значения вызывающей программой. Таким образом возвращаемое значение может использовать как r-значение, но не как l-значение.

```
class Point
{ int x, y;
public:
// . . . setPoint(), getX() и т. д.
const Point* closestPointPtr(const Point& p)
  { return (x*x + y*y < p.x*p.x + p.y*p.y) ? this : &p; }              // OK
const Point* closestPointRef(const Point& p)
  { return (x*x + y*y < p.x*p.x + p.y*p.y) ? *this : p; } } ;          // OK
```

Теперь клиент ограничен в использовании объектов.

```
Point p1, p2; p1.setPoint(20,40); p2.setPoint(30,50);
Point *ptr = p1.closestPointPtr(p2);          // синтаксическая ошибка:
                                              // должно быть const
Point &ref = p1.closestPointRef(p2);          // синтаксическая ошибка:
                                              // должно быть const
const Point *p = p1.cloststPointPtr(p2);      // *p - r-значение
p->setPoint(0,0);                             // синтаксическая ошибка:
                                              // нельзя изменять объект
const Point &r = p1.closestPointRef(p2);      // r не может быть l-значением
r.setPoint(10,10);                            // синтаксическая ошибка:
                                              // нельзя изменять объект
```

Так для чего же хорошо использовать указатель p и ссылку r? Ответ очевиден: они не позволяют вызывать функции вида setPoint(), модифицирующие целевой объект, но должны иметь возможность вызывать функции типа getX(), этот объект не изменяющие. Примерно так.

```
int x1 = p->getX();        // p указывает на константу Point
int x2 = r.getX();         // r ссылается на константу Point
```

Если вам нравится такая цепочка, можно разделить указатель и ссылку и получить координаты ближайшей точки таким образом:

```
x1=(p1.closestPointPtr(p2)).getX();          // синтаксическая ошибка
x2=p1.closestPointPtr(p2).getX();            // синтаксическая ошибка
```

Надеюсь, что, еще не владея бегло деталями этого синтаксиса, вы уловили общий смысл обсуждения. C++ предлагает ключевое слово const, которое используется компилятором и сопровождающим приложение программистом для определения изменения объекта при выполнении программы. Мы рассмотрели способ предотвращения изменений в значении, указателе, указателе-параметре и возвращаемом значении функции (указателе или ссылке).

Естественно, указатель или ссылка на объект-константу не могут изменяться при вызове таких функций, как setPoint(), ведь setPoint() изменяет состояние объекта по указателю или ссылке. Но что плохого в вызове такой невинной функции, как getX()? Она же не изменяет состояния объекта по указателю или ссылке?

Это снова возвращает нас к фундаментальному идеологическому вопросу, обсуждаемому в главе 7 при описании параметров функции. Откуда мы знаем, изменяет функция параметр или нет? Мы не хотим исследовать для этого тело функции, а хотим взглянуть просто на ее заголовок. Если в заголовке говорится, что параметр есть константа, то понятно, что он не изменяется. Если же в заголовке это не сказано, можно считать, что он изменятся, независимо от того, что делает функция.

Компилятор C++ следует той же логике. Он достаточно интеллектуален, чтобы найти ключевое слово const в заголовке и пометить изменения параметра как синтаксическую ошибку. Но он не настолько умен, чтобы анализировать исходный код функции и приходить к независимому заключению об изменениях параметров. Предполагается, что если ключевое слово const отсутствует, то параметры изменяются.

Теперь вернемся к двум функциям setPoint() и getX(). Откуда известно, что первая изменяет объект, а вторая — нет? Достаточно взглянуть на программу и заголовок. Это очевидно, не так ли? Но не для компилятора C++. Он помечает вызов setPoint() как ошибочный не потому, что знает об изменении объекта в данной функции, а потому, что не видит свидетельства обратного. Для компилятора функция getX() ничем не лучше setPoint(). Если ничто не указывает, что getX() оставляет объект без изменений, то компилятор приходит к выводу, что getX() изменяет состояние объекта.

Вот здесь-то в C++ используется четвертый смысл ключевого слова const. Это ключевое слово включается между закрывающей круглой скобкой списка параметров и открывающей фигурной скобкой тела функции. В прототипе функции оно вставляется между закрывающей круглой скобкой и точкой с запятой. Здесь класс Point явно указывает, что его функции-члены делают:

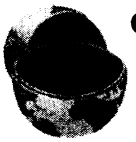
1) с параметрами функции;

2) с возвращаемыми функцией значениями;

3) с элементами данных целевого объекта:

```
class Point
{ int x, y;
public:
  void setPoint(int a, int b)     // модифицирует поля, не так ли?
    { x = a; y = b; }
  int getX() const                // не модифицирует поля: где свидетельство?
    { return x; }
  int getY() const                // не модифицирует поля: где свидетельство?
    { return y; }
  const Point& closestPointRef(const Point& p) const              // красиво
    { return (x*x + y*y < p.x*p.x + p.y*p.y) ? *this : p; }} ;     // OK
```

Действительно красиво, правда? Обсуждение было достаточно запутанным, но ключевое слово const имеет в C++ различные смыслы, с этим ничего не поделаешь. По крайней мере еще один будет обсуждаться в следующем разделе. Так что данное ключевое слово нужно принимать всерьез. При написании серверной части это ваш основной инструмент для передачи своих идей другим людям, а при чтении программы — важное средство, помогающее понять намерения разработчика. Используйте ключевое слово const там, где это возможно. Если вы не сообщите о своих замыслах (в плане операций функций-членов) программисту, занимающемуся клиентской частью или сопровождением приложения, это будет серьезной ошибкой.



Советуем Применяйте ключевое слово const для значения (или указателя), не модифицируемого после инициализации. Используйте его, когда нужно сообщить, что параметр функции (или указатель) не изменяется при выполнении функции, когда возвращаемое функцией значение (указатель или ссылка) не изменяются в клиенте, когда функция-член при передаче сообщения не изменяет состояния целевого объекта. При изучении исходного кода C++ также анализируйте применение ключевого слова const. Не спешите делать выводы, кажущиеся очевидными.

Итак, нужно обращать на ключевое слово const самое пристальное внимание. Это один из моментов, с которого начинается программирование на C++.

Статические компоненты класса

В данном разделе обобщается понятие элементов данных класса. Концептуально класс представляет собой план объекта. Спецификация класса описывает, что включает в себя каждый объект класса: данные и функции.

При создании экземпляра объекта класса для него формируется отдельный набор элементов данных. Это происходит независимо от того, как создается объект: через определение объекта как локальной или глобальной именованной переменной, с помощью операции new или неименованной динамической переменной, посредством передачи объекта по значению в параметре функции или возврата объекта по значению из функции. Каждый экземпляр объекта имеет собственный набор значений элементов данных: private, public или protected.

Между тем, нет никакой необходимости создавать для каждого объекта отдельный набор функций-членов. Для каждой функции объектный код генерируется только один раз. Кроме параметров, предусмотренных программистом, каждая функция-член имеет неявный параметр — указатель на целевой объект. Когда в ее вызове в качестве цели указывается конкретный объект, функции передается указатель this на целевой объект. В результате она работает с элементами данных целевого объекта.

Применение глобальных переменных как характеристик класса

Иногда логичнее и эффективнее с точки зрения использования памяти реализовать для всех объектов класса одну общую копию элементов данных, а не поддерживать для каждого объекта индивидуальные копии.

Например, приложению может потребоваться счетчик экземпляров класса. Рассмотрим класс Point, содержащий элемент данных count. Логически этот элемент данных принадлежит к классу точно так же, как и любые другие.

```
class Point {
    int x, y;           // индивидуальные для каждого объекта Point
    int count;          // общий для все объектов Point
  . . . } ;
```

С практической точки зрения здесь есть ряд проблем. Нужен только один счетчик точек. Если приложение создает тысячи объектов Point, нет смысла тысячекратно дублировать поля count и поддерживать в каждом поле одно и то же значение. Кроме того, как поддерживать такой счетчик count? Его следует увеличивать при каждом создании нового объекта Point. Хорошее место для этого — конструктор Point. Аналогично деструктор будет хорошим местом для уменьшения счетчика count.

```
Point::Point (int a, int b)          // общий конструктор
  { x = a; y = b; count++; }         // увеличение счетчика объектов
```

Но это нельзя назвать хорошим решением. Увеличивается только один счетчик во вновь созданном объекте, а элементы данных, принадлежащие другим объектам, не увеличиваются. Кроме того, поле count создаваемого объекта не инициализируется предыдущим значением поля данных count в ранее созданных объектах. Следовательно, данный конструктор увеличивает неинициализированное значение. Действительно, хорошего в такой версии мало.

Для решения задачи можно было бы использовать глобальную переменную, инициализируя ее нулем в начале выполнения программы, а затем увеличивая (в конструкторе) при создании очередного объекта и уменьшая (в деструкторе) при уничтожении объекта.

Например, глобальная переменная могла бы подсчитывать число созданных экземпляров точек: конструктор будет увеличивать счетчик, а деструктор — уменьшать его. В листинге 9.5 показан пример реализации класса Point, где применяется такой подход. Конструктор Point может использоваться как конструктор по умолчанию (клиент не передает аргументов), конструктор преобразования (клиент передает один аргумент) и общий конструктор (клиент передает два аргумента — координаты точки). Для иллюстрации в конструктор и деструктор включены операторы отладки, так что можно следить за порядком вызовов функций. Функция quantity() возвращает значение count, поэтому клиент не нужно будет менять при изменении имени глобальной переменной. Переменная count явным образом инициализируется нулевым значением. Согласно правилам языка C++, ее можно инициализировать нулем и неявно, но явная инициализация предпочтительнее.

Листинг 9.5. Использование глобальной переменной для подсчета создаваемых объектов

```
#include <iostream>
using namespace std;

int count = 0;      // откуда при сопровождении программы известно, что это принадлежит Point?

class Point {
     int x, y;                                  // частные координаты
   public:
     Point (int a=0, int b=0)                   // общий конструктор
       { x = a; y = b; count++;
         cout << " Создана точка: x=" << x << " y=" y << endl; }
     void set (int a, int b)                    // функция-модификатор
       { x = a; y = b; }
     void get (int& a, int& b) const            // функция-селектор
       { x = a; y = b; }
     void move (int a, int b)                   // функция-модификатор
       { x += a; y += b; }
     ~Point()
       { count-;
         cout << " Удалена точка: x=" << x << " y="<< y << endl; }
   } ;

int quantity()                                  // доступ к глобальной переменной
   { return count; }

int main()
{  cout << "Число точек: " << quantity() << endl;
   Point *p = new Point(80,90);                 // динамически распределяемый объект
   Point p1, p2(30), p3(50,70);                 // начало координат, ось x, общая точка
   cout << "Число точек: " << quantity() << endl;
   return 0;                                    // динамический объект не удаляется должным образом
}
```

```
Число точек: 0
 Создана точка: x=80 y=90
 Создана точка: x=0 y=0
 Создана точка: x=30 y=0
 Создана точка: x=50 y=70
Число точек: 4
 Удалена точка: x=50 y=70
 Удалена точка: x=30 y=0
 Удалена точка: x=0 y=0
```

Рис. 9.6. *Результат программы из листинга 9.5*

Результаты выполнения программы показаны на рис. 9.6. Как видно, все начинается с создания неименованного объекта Point. Первая создаваемая именованная точка (p1) инициализируется конструктором по умолчанию, p2 инициализируется конструктором преобразования, а p3 — общим конструктором. Эти именованные переменные Point уничтожаются в обратном порядке. Обратите внимание, что здесь не удаляется должным образом неименованная динамическая переменная — объект Point. В результатах программы не видно сообщения, свидетельствующего о ее уничтожении.

Такая конструкция работает, но имеет ряд недостатков, затрудняющих ее преобразование в большую программу. Любая часть клиента может обращаться к переменной count и модифицировать ее значение. Тем самым создаются зависимости между отдельными частями программы. Имя глобальной переменной может конфликтовать с другими глобальными именами проекта или с библиотечными именами. Каждый участвующий в проекте разработчик должен быть осведомлен о данном имени, хотя большинству из них даже не нужно знать о классе Point. Тем самым без какой бы то ни было необходимости расширяется область знания программистов о разных частях проекта, увеличивается сложность его разработки и сопровождения.

Между тем, основная проблема данного решения в том, что в нем знание разработчика не передается сопровождающему приложение программисту. При определении переменной count известно, что эта переменная должна подсчитывать число объектов Point, а не число объектов Rectangle или что-то еще, однако в синтаксисе ничто не указывает, что данная переменная ассоциирована с конкретным классом. При сопровождении программы узнавать об этом придется из комментариев (а они могут быть непонятными, устаревшими или вовсе отсутствовать) или путем исследования больших сегментов программного кода.

Четвертый смысл ключевого слова static

В C++ можно решить данную проблему, применив слово static еще для одной цели. Как рассказывалось в главе 6, это ключевое слово может иметь разный смысл.

Во-первых, оно обозначает глобальную переменную, которая видима только в функциях, определенных в том же исходном файле. Такую переменную нельзя сделать доступной в другом файле, объявив ее там как extern.

Во-вторых, данное ключевое слово применяется к переменной, значение которой не исчезает при завершении работы функции (как значения других локальных переменных), а сохраняется системой и может использоваться для инициализации переменной, если функция вызывается снова. В-третьих, это ключевое слово применяется к функции, которая может вызываться только из того же файла.

В данном разделе ключевое слово static применяется к элементам данных класса. Оно означает как раз то, что нам нужно — для всех объектов класса существует только один экземпляр таких данных. Данные будут общими для всех объектов типа класса.

Во всех других отношениях статические элементы данных — это обычные данные. При необходимости они могут определяться как public, private или protected. Синтаксис определения статических элементов данных и доступа к ним такой же, как и у других элементов данных. Единственная разница — в ключевом слове static.

```
class Point {
    int x, y;                          // закрытые координаты
    static int count;                  // еще один смысл этого ключевого слова
  public:
    Point (int a=0, int b=0)           // многосторонний конструктор
       { x = a; y = b; count++; }
```

```
void set (int a, int b)              // функция-модификатор
   { x = a; y = b; }
void get (int& a, int& b) const      // функция-селектор
   { x = a; y = b; }
void move (int a, int b)             // функция-модификатор
   { x += a; y += b; }
~Point()                             // деструктор
   { count-; }} ;
```

Здесь элемент данных count — это единое совместно используемое значение, доступное для всех экземпляров объекта Point. Этот элемент данных находится в той же области действия, где и определение класса, и доступен в ней.

Инициализация статических элементов данных

Подобно глобальным переменным, не являющимся элементами данных, статические элементы данных инициализируются вне спецификации класса. В отличие от глобальной переменной count, статический элемент данных count является компонентом класса и должен инициализироваться явным образом. Для элементов данных (статических или нет) неявной инициализации не предусмотрено. Чтобы использовать имя любого компонента класса вне его фигурных скобок, нужно применять операцию области действия класса, показывающую, к какому классу относятся эти данные.

```
int Point::count = 0;          // это не присваивание (видите здесь имя типа?)
```

Выше уже говорилось о различии между присваиванием и инициализацией в C++. Часто операция присваивания может обозначать или присваивание, или инициализацию. Инициализация распознается по наличию рядом с именем переменной имени типа. Здесь разница между инициализацией и присваиванием очень важна. Инициализация вполне законна и имеет один и тот же синтаксис для общедоступных и не общедоступных статических элементов данных класса. Для статических элементов данных, не являющихся общедоступными, присваивание незаконно.

```
Point::count = 0;     // присваивание (незаконно для закрытых данных 'count')
```

Для статических элементов данных допускается только один оператор инициализации. Следовательно, этот оператор нужно помещать в файл реализации .cpp вместе с определениями функций-членов, но не в заголовочный файл класса.

Доступ к статическим элементам данных идентичен доступу к нестатическим элементам данных. Функции, не являющиеся членами класса (например, функция quantity()), не могут обращаться к закрытым статическим элементам данных непосредственно. Чтобы избежать этой проблемы, можно оформить функцию quantity() как член класса:

```
class Point {
    int x, y;                        // закрытые координаты
    static int count;                // еще один смысл этого ключевого слова
  public:
    Point (int a=0, int b=0)         // многосторонний конструктор
    { x = a; y = b; count++; }
    int quantity() const             // не изменяет состояние объекта
    { return count; }
  . . . } ;
```

Обратите внимание, что предыдущая (глобальная) версия quantity()не имела модификатора const. Лишь функция-член может обещать не изменять элементы данных своего целевого объекта. Глобальные функции целевого объекта не имеют.

Статические элементы данных не могут быть компонентом объединения (union). Они не могут быть битовым полем класса. И объединения, и битовые поля обозначают специальное использование памяти, принадлежащей конкретному объекту. Статические элементы данных принадлежат не конкретному объекту, а классу в целом. Вероятно, вам не потребуется часто использовать объединения и битовые поля или определять их как статические, тогда такое ограничение особо не повредит.

Статические функции-члены

Вот почти и все, но прежде чем завершить данную главу, нужно познакомиться с пятым смыслом ключевого слова static в C++. Это ключевое слово может применяться как модификатор в функции-члене класса, не обращающейся к нестатическим элементам данных. Другими словами, такая статическая функция-член может обращаться только к своим параметрам, статическим элементам данных и (о, ужас!) к глобальным переменным.

Хорошим кандидатом на превращение функции-члена в статическую является функция quantity(). Она не имеет никаких параметров, обращается к элементу данных count и не получает доступа ни к каким нестатическим элементам данных.

```
class Point {
    int x, y;                                  // закрытые координаты
    static int count;                          // закрытый счетчик объектов
  public:
    Point (int a=0, int b=0)                   // многосторонний конструктор
    { x = a; y = b; count++; }
    static int quantity()                      // не изменяет состояние объекта
    { return count; }                          // это не может быть const
  . . . } ;
```

Статическая функция не может объявляться как const, даже если она не изменяет тех значений, к которым обращается. Статическая функция-член обращается к параметрам, статическим элементам данных и глобальным переменным. Данные значения не являются частью состояния объекта. В применении к функции-члену ключевое слово const утверждает, что она не изменяет элементы данных целевого объекта, к которым эта функция обращается. Поскольку статическая функция не обращается к нестатическим элементам данных, ничего обещать нельзя.

Понятно, что это звучит не очень убедительно, но за всем сказанным стоит весьма жесткая логика. Приведенная выше серия примеров начиналась с функции quantity(), реализованной как глобальная функция, обращающаяся к глобальной переменной count. Затем глобальная переменная count была превращена в статические компонентные данные. Наконец, функция quantity() в соответствии с переменной count стала функцией-членом класса Point. Как нестатическая функция-член, она была определена как const, показывая тем самым, что она не изменяет нестатические элементы данных класса. Наконец, эта функция была превращена в статическую функцию-член, и модификатор const стал нерелевантным.

Подобно статическим элементам данных и аналогично вызову нестатической функции, статическая функция-член может вызываться через целевой объект класса (или указатель на объект класса). Ее можно вызывать непосредственно с помощью операции области действия класса, даже если объект класса создан не был.

```
int main()
{ cout << "\nЧисло точек " << Point::quantity();          // выводит 0
  Point p1(20,40);
  cout << "\nЧисло точек " << p1::quantity();            // выводит 1
  cout << "\nЧисло точек " << Point::quantity();         // выводит 0
. . . }
```

В листинге 9.6 приведен пример класса Point со статическим элементом данных count. Он инициализируется вне определения класса, хотя является закрытым. Функция quantity() определяется как static, и обращаться к ней можно с помощью операции области действия (первый вызов) и целевого объекта (второй вызов).

Листинг 9.6. Использование статических элементов данных и функций-членов

```
#include <iostream>
using namespace std;

class Point {
     int x, y;                                        // закрытые координаты
     static int count;
  public:
     Point (int a=0, int b=0)                         // общий конструктор
       { x = a; y = b; count++;
         count << " Создана точка: x=" << x << " y=" y << endl; }
       static int quantity()                          // const не разрешается
            { return count; }
       void set (int a, int b)                        // функция-модификатор
       { x = a; y = b; }
       void get (int& a, int& b) const                // функция-селектор
       { x = a; y = b; }
       void move (int a, int b)                       // функция-модификатор
       { x += a; y += b; }
       ~Point()                                       // деструктор
       { count-;
         cout << " Удалена точка: x=" << x << " y="<< y << endl; }
  } ;

int Point::cont = 0;

int main()
{ cout << "Число точек: " << Point::quantity() << endl;
  Point p1, p2(30), p3(50,70);                        // начало координат, ось x, общая точка
  cout << "Число точек: " << p1. quantity() << endl;
  return 0;
  }
```

Результат выполнения программы показан на рис. 9.7.

Рассмотрим еще одну версию класса Point, содержащего функцию, сравнивающую координаты своих двух параметров Point и возвращающую true, если координаты совпадают.

```
class Point {
     int x, y;                  // закрытые координаты
     static int count;          // закрытый счетчик объектов
  public:
     Point (int a=0, int b=0)   // многосторонний конструктор
       { x = a; y = b; count++; }
     static int quantity()      // const не разрешается
     { return count; }
     bool samePoints (const Point &p1, const Point &p2)
       { return p1.x == p2.x && p1.y == p2.y }
  . . . } ;
```

```
Создана точка: x=640 y=0
Число точек: 1
Создана точка: x=0 y=0
Создана точка: x=30 y=0
Создана точка: x=50 y=70
Число точек: 4
Удалена точка: x=50 y=70
Удалена точка: x=30 y=0
Удалена точка: x=0 y=0
```

Рис. 9.7. *Результат программы из листинга 9.7*

Переданы ли здесь идеи разработчика сопровождающему приложение программисту? Нет. Например, очевидным промахом является то, что в данной версии не отражается тот факт, что функция samePoints() не изменяет состояние целевого объекта. Следовательно, ее нужно определить как const. Но это еще не все. Нужно указать также, что эта функция работает только со своими параметрами. Она может быть глобальной, так как не нуждается в элементах данных целевого объекта Point и работает только с элементами данных, переданными ей в параметрах. Между тем она включена в класс Point. Это показывает, что она логически принадлежит классу Point и имеет дело с объектами данного класса, а не с объектами класса Rectangle, Circle или др. Вот почему ее следует определить как static.

Для иллюстрации посмотрим, как можно вызывать эту функцию. Вот несколько способов:

```
Point p1, p2(30), p3(50,70);
if (p1.samePoints(p2,p3)==true) cout << "Точки совпадают\n";
```

Но какое отношение объект p1 имеет к сравнению p2 и p3? Помните байку про крокодила и обезьяну? Некрасивый вариант. Еще один способ — использовать объект p2 дважды:

```
Point p1, p2(30), p3(50,70);
if (p2.samePoints(p2,p3)==true) cout << "Точки совпадают\n";
```

Все равно некрасиво. Объект должен использоваться только один раз. Давайте определим эту функцию как статическую:

```
class Point {
    int x, y;                                   // частные координаты
    static int count;                           // частный счетчик объектов
  public:
    Point (int a=0, int b=0)                    // многосторонний конструктор
       { x = a; y = b; count++;}
    static int quantity()                       // const не разрешается
    { return count; }
    static bool samePoints (const Point &p1, const Point &p2)
       { return p1.x == p2.x && p1.y == p2.y; }
  . . . } ;
```

Теперь можно вызывать функцию с помощью операции области действия:

```
Point p1, p2(30), p3(50,70);
if (Point::samePoints(p2,p3)==true) cout << "Точки совпадают\n";
```

Отлично!

Так когда же использовать статические элементы данных и статические функции? Если элементы данных определяются как статические, это показывает, что такие глобальные данные логически принадлежат классу (пример — count). Когда в качестве статических определяются функции-члены, это свидетельствует о том, что глобальные функции логически принадлежат классу и работают с его статическими данными, глобальными данными или параметрами, но не с нестатическими данными (примеры — quantity(), samePoint()).

Применение статических данных и функций не является при изучении C++ главной задачей, однако нужно понимать эти вопросы и использовать ключевое слово static для лучшей передачи своих намерений программистам, отвечающим за клиентскую часть и сопровождение приложения.

Итоги

В этой главе рассматривалось использование классов C++ как инструментов для создания программ. Применение классов устраняет недостатки, свойственные глобальным функциям как инструменту объектно-ориентированного программирования.

Первый недостаток глобальных функций состоит в следующем: они не всегда сообщают, что имел в виду разработчик и что функции, обращающиеся к одной и той же структуре данных, на самом деле связаны логически.

Например, если программа использует структуры данных `Point` и `Rectangle`, а разработчик помещает все функции доступа к `Point` в одно место, а все функции доступа к `Rectange` — в другое, то все замечательно. Если же функции доступа разделены, компилятор жаловаться не будет, но при сопровождении в программе сложно разобраться. Важна дисциплина программирования.

Второй недостаток в том, что при применении глобальных функций инкапсуляция — дело сугубо добровольное. Программист может отказаться от функций доступа и обращаться непосредственно к полям структуры. Правила языка этого не запрещают. Такой подход также требует дисциплины от программиста.

Третий недостаток — глобальная природа функций. Их имена являются частью глобального пространства имен и могут конфликтовать с другими именами функций. Поэтому чтобы избежать конфликтов, программисту нужно знать обо всех применяемых в проекте именах функций, а это увеличивает сложность разработки системы.

Классы C++ позволяют устранить перечисленные недостатки. Объединяя вместе данные и операции с ними, мы избавляемся от первого недостатка. Возможность контролировать доступ к полям данных устраняет второй недостаток, а использование области действия класса — третий.

Классы несут в себе большой потенциал для улучшения качества ПО.

Глава 10

Операторные функции

Темы данной главы

- ✔ Перегрузка операций
- ✔ Ограничение на перегрузку операций
- ✔ Перегруженные операции как компоненты класса
- ✔ Учебный пример: рациональные числа
- ✔ Смешанные типы как параметры
- ✔ Дружественные функции
- ✔ Итоги

В предыдущей главе рассказывалось о синтаксисе и семантике классов языка C++. Язык C++ — не первый язык программирования, поддерживающий концепцию классов, но именно в нем это впервые успешно реализовано в "индустриальном масштабе".

Первоначально C++ внедрялся достаточно медленно, так как в индустрии ПО с сомнением воспринималась его эффективность и надежность. Сомнения в основном были безосновательны. Большинство программ C++ занимают столько же памяти, сколько эквивалентные программы на C, не более, и почти все выполняются не медленнее, чем эквивалентные программы на языке C. Безусловно, есть некоторые исключения, связанные с использованием библиотеки `iostream`, виртуальных функций и шаблонов. (О них подробнее рассказывается в следующих главах.) Между тем значительный прогресс в области создания аппаратных средств привел к увеличению объемов памяти и быстродействия компьютеров. Это ослабило требования к памяти и к скорости выполнения программ. Опыт применения языка C++ ясно демонстрирует, что программирование с применением классов может быть весьма эффективным. Ожидается, что все новые языки программирования будут поддерживать классы.

Недоверие к надежности языка еще полностью не исчезло. Опыт применения C++ подтвердил, что существуют опасности и определенные недостатки, которых программистам следует избегать. Однако это не стало препятствием на пути превращения C++ в основной язык программирования широкого спектра приложений, только сложность этого языка является главным препятствием для достижения надежности. Но, как было показано в предыдущей главе, в основе классов C++ лежит достаточно простая идея. Классы C++ помогают программистам:

- Связывать данные и операции объекта
- Управлять внешним доступом к элементам класса

- Вводить дополнительные области действия для предотвращения конфликтов имен
- Переносить обязанности с клиентов на серверы

В предыдущей главе показано, что разработчик C++ Бьерн Страуструп включил в классы C++ намного больше, чем предусматривает приведенный короткий список. Конструкторы и деструкторы помогают объектам класса управлять своими ресурсами (в основном динамически распределяемой памятью). Доступность этих функций возлагает дополнительное бремя на разработчика классов (программиста, отвечающего за серверную часть приложения). Он обязан предусмотреть разнообразные конструкторы для поддержки клиента в разнообразных контекстах. Кроме того, программист должен обеспечить инициализацию данных объекта, но это уже незначительный побочный эффект.

Использование составных объектов еще более усложняет дело. Разработчику класса-контейнера надо позаботиться об инициализации компонентов объекта. Новый синтаксис для этого реализует список инициализации компонентов. Идея составного класса требует использования таких дополнительных деталей, как компоненты-константы, компоненты-указатели и рекурсивные компоненты. Принцип атрибутов класса ведет к другим расширениям этой идеи, таким, как статические элементы данных и статические функции, характеризующие класс в целом, а не отдельные экземпляры объектов класса.

Как уже упоминалось, при разработке C++ ставилась еще одна цель: интерпретировать экземпляры класса так же, как переменные встроенных типов. В предыдущей главе этот принцип проявлял себя в виде единообразного синтаксиса инициализации и объектов, и переменных. В данной главе мы обсудим еще одно проявление той же идеи — ее расширение на операции C++, в результате чего один и тот же синтаксис выражений с операциями можно применять как к объектам классов, так и к переменным встроенных типов в обычных выражениях C++.

Обычно C++ позволяет делать это несколькими способами. Мы обсудим разные методы реализации перегруженных операторных функций. Данные методы помогают эффективнее использовать C++, а также позволяют лучше понять внутренние механизмы программы C++.

Перегрузка операций

В C++ концепция определяемых программистом типов (классов) является расширением концепции встроенных числовых типов. Здесь можно определять переменные новых типов, применяя тот же синтаксис, что и для простых числовых переменных. Подобно встроенным типам, разрешается использовать экземпляры объектов определяемых программистом типов как элементы массива или как элементы данных еще более сложных типов, передавать объекты таких вновь образованных типов, как параметры и, возвращать их из функций, присваивать указателям и ссылкам определяемые программистом значения, применяя такой же синтаксис, как для встроенных значений. С помощью того же синтаксиса, что и для встроенных типов, можно также определять указатели и ссылки на значения-константы.

Такое подобие не случайно. Одна из важных целей разработки C++ заключалась в интерпретации определяемых программистом типов таким же образом, как встроенных типов. Данная цель не имеет ничего общего с объектно-ориентированным программированием, улучшающим продуктивность разработки, повышающим эффективность сопровождения ПО или положительно отражающимся на других аспектах разработки ПО. Это чисто эстетическая цель. И вполне законная. Программирование, как и любая творческая деятельность, имеет значительную эстетическую составляющую. Хотя в книгах по программированию данная тема обсуждается редко, создаваемые программы должны быть настолько же элегантны, насколько удобны в чтении и сопровождении.

Конечно, многие программы, особенно крупные, трудно назвать элегантными. Как, впрочем, удобными в сопровождении, легко читаемыми и переносимыми. Между тем язык спроектирован так, чтобы программист имел возможность достичь этих целей.

В то же время существует значительный разрыв между одинаковой интерпретацией классов и числовых типов. Определяемые программистами типы C++ не являются в точности такими же, как "естественные" числовые типы. Самая большая разница в том, что операции C++ — сложение, вычитание, сравнение на равенство и т. д. — нельзя применять к объектам определяемых программистом типов. Конечно, можно написать функции для реализации подобных операций, но запись выражений при этом будет весьма сложной.

Рассмотрим простой пример: комплексные числа, характеризуемые значениями действительной и мнимой части. Те, кто не знаком с мнимыми числами, могут рассматривать их как точки на плоскости, где действительные компоненты соответствуют координате x, а мнимые — координате y. При сложении или вычитании сложных чисел результатом также будет мнимое число. Его действительная часть есть результат сложения (или вычитания) действительных компонентов двух операндов, а мнимая — результат сложения (или вычитания) мнимых компонентов. Умножение и деление выполняется сложнее, но тоже реализуются как операции с компонентами комплексного числа.

Представим комплексные числа как класс с двумя элементами данных — `real` и `imag`. Для простоты обсуждения объявим их `public` (в `private` они превратятся в следующей версии класса).

```
struct Complex {
  double real, imag; } ;              // общедоступные элементы данных
```

В листинге 10.1 дан пример программы, определяющей экземпляры типа `Complex`, инициализирующие и выполняющие некоторые арифметические операции с объектами.



Внимание Данный пример нельзя считать хорошим примером программы C++. Авторы большинства книг по C++ избегают демонстрировать отрицательные примеры. В результате читателю не удается научиться отличать хорошие программы C++ от плохих. Это все равно, что пытаться научиться живописи, только посещая музеи, в которых представлены шедевры. Программирование на C++ — всегда попытка найти лучшее решение, превосходящее конкурирующий или альтернативный вариант. Вместо демонстрации правильного решения я предпочитаю показывать ущербное, пояснять, что в нем неверно и как можно его поправить. А затем привожу лучшее решение и рассказываю о том, чем именно оно лучше.

В примере (см. листинг 10.1) клиент выполняет вычисления с комплексными числами, используя прямой доступ к общедоступным компонентам объекта. В клиенте вместо функций-членов определяются имена и элементы данных `real` и `imag`. В результате клиентская программа представляет собой сочетание доступа к полям данных и вычисления с полями данных. Смысл этих вычислений не выражается в вызовах функций, его трудно уяснить из анализа деталей вычислений на нижнем уровне. Операции низкого уровня не перенесены на функции-серверы, и программисту придется держать в уме одновременно несколько уровней алгоритма: вычисления высокого уровня и детальные операции. Обязанности не разделены, и изменения в конструкции класса `Complex` повлияют на всю программу. Применять ключевое слово `struct` вместо `class` здесь не стоит, так как все элементы данных общедоступны.

Листинг 10.1. Пример операций с объектами класса `Complex`

```
#include <iostream>
using namespace std;

struct Complex {                                    // тип, определяемый программистом
     double real, imag; } ;

int main()
{   Complex x, y, z1, z2;                           // объекты типа Complex
    x.real = 20; x.imag = 40;                       // инициализация
    y.real = 30; y.imag = 50;
    cout << "Первое значение: ";
    cout << "(" << x.real << ", " << x.imag << ")" << endl;
    cout << "Второе значение: ";
    cout << "(" << y.real << ", " << y.imag << ")" << endl;
    z1.real = x.real + y.real;                      // сложить действительные компоненты в z1
    z1.imag = x.imag + y.imag;                      // сложить мнимые компоненты
    cout << "Сумма двух значений: ";
    cout << "(" << z1.real << ", " << z1.imag << ")" << endl;
    z2.real = x.real + y.real;                      // сложить действительные компоненты в z2
    z2.imag = x.imag + y.imag;                      // сложить мнимые компоненты
    z1.real = z.real + x.real;                      // сложить действительные компоненты в z1
    z1.imag = z.imag + x.imag;                      // сложить мнимые компоненты
    cout << "Сложение первого значения с суммой: ";
    cout << "(" << z1.real << ", " << z1.imag << ")" << endl;
    z2.real += 30.0;                                // сложение с действительной частью z2
    cout << "Сложение 30 с суммой: ";
    cout << "(" << z2.real << ", " << z2.imag << ")" << endl;
    return 0;
  }
```

```
Первое значение:  <20, 40>
Второе значение:  <30, 50>
Сумма двух значений:                  <50, 90>
Сложение первого значения с суммой: <70, 130>
Сложение 30 с суммой:                 <80, 90>
```

Рис. 10.1. *Результат выполнения программы из листинга 10.1*

Результат выполнения данной программы показан на рис. 10.1. Хотя этот пример нельзя считать хорошим, но мы получили неплохую отправную точку для обсуждения перегрузки операторных функций. Кроме того, хочется воспользоваться случаем, чтобы повторить список случаев плохого применения C++. Это очень важный список: повторяйте его при анализе своей программы. Вы сможете оценить, насколько корректно используете C++, и улучшить качество создаваемого ПО.

Чтобы инкапсулировать детали реализации и изолировать от них клиент, нужно написать функции доступа, манипулирующие объектами типа `Complex` для обслуживания клиента. Например, если требуется добавить переменные данного типа, нужно написать функцию с двумя параметрами типа `Complex`, которая выполняет необходимые операции с компонентами объекта и возвращает результат того же типа. Это означает, что интерфейс функции, например `addComplex()`, может выглядеть следующим образом:

```
Complex addComplex(const Complex &a, const Complex &b);
```

Как уже упоминалось выше, сложение двух комплексных чисел требует отдельного сложения их вещественных и мнимых компонентов.

```
Complex addCompex(const Complex &a, const Complex &b)
{ Complex c;                          // локальный объект
  c.real = a.real + b.real;           // сложение вещественных компонентов
```

```
c.imag = a. imag + b.imag;              // сложение мнимых компонентов
return c; }
```

Чтобы использовать данную функцию, в клиенте определяются и инициализируются переменные типы Complex. Клиент передает их этой функции как параметры и использует возвращаемое значение типа Complex.

```
Complex x, y, z1, z2;                   // объекты типа Complex
x.real = 20; x.imag = 40;               // инициализация
y.real = 30; y.imag = 50;
z1 = addComplex(x,y);                   // использование в вызове функции
```

Все это очень здорово (и весьма тривиально). Большинство программистов применяют подобный функциональный стиль программирования, не чувствуя, что использование таких имен функций, как addComplex(), делает их программы трудночитаемыми. Настоящий программист (во всяком случае, считающий себя таковым) напишет код клиента так:

```
Complex x, y, z1, z2;                   // объекты типа Complex
x.real = 20; x.imag = 40;               // инициализация
y.real = 30; y.imag = 50;
z2 = x + y;                             // использование в выражении
```

Если это сделать, то компилятор сообщит, что операция сложения не определена, несмотря на все претензии C++ на универсальный подход и попытки одинаково интерпретировать разные типы. Поскольку использовать встроенные операции для определяемых программистом типов нельзя, придется придумать новые функции вида addComplex() и реализовать с их помощью необходимые операции. Неравноправие между определяемыми типами и встроенными типами C++ болезненно воспринимается каждым настоящим программистом.

В качестве лекарства C++ предлагает следующее. Программист может ограничиться специальными именами функций с ключевым словом operator и знаком операции, которую нужно использовать в исходном коде, например +. Он разрабатывает и реализует такую функцию operator+() точно так же, как реализуется любая другая функция, в частности addComplex(). Как язык, поддерживающий программиста, C++ позволяет вызывать данную функцию с помощью знака операции, соответствующего знаку операции в имени функции. Если функция называется operator+(), то можно вызвать ее с помощью такой же записи, как для встроенных числовых типов:

```
z = x + y;          // на самом деле это z = operator+(x,y);
```

Ну не замечательно ли? Предлагаемый функцией сервис ассоциируется со встроенной операцией C++. Великолепно!

Но это не столь уж уникальное средство. В C++ одно и то же имя функции в одной и той же области действия может представлять разные алгоритмы — должны лишь различаться сигнатуры функций (см. главу 7, где о перегрузке имен функций рассказывается подробнее). Когда клиент вызывает данную функцию, компилятор сопоставляет типы фактических аргументов с объявлениями функции в этой области действия и определяет, какое из них нужно использовать для реализации вызова функции.

Сказанное относится к любому имени функции в C++. Что касается арифметических операций, то перегрузка операций применяется в любом языке программирования, а не только в C++. Она означает множественную интерпретацию одного и того же символа. Рассмотрим, например, операцию сложения:

```
int a,b,c;
float d,e,f;
a = 20, b = 30; d = 40.0; e = 50.0;
c = a + b; f = d + e;                   // разные операции, один символ
```

В C++ (и в других языках) операция + используется для сложения целых чисел или чисел с плавающей точкой, а это совершенно разные действия. Для целых переменных каждый бит одного операнда складывается с каждым битом другого и с битом переноса из бита нижнего порядка.

Для значений с плавающей точкой двоичное представление состоит из мантиссы и экспоненты. Чтобы уйти от сложности двоичных (или шестнадцатеричных) значений, проиллюстрируем данный вопрос с помощью десятичной системы. В представлении с мантиссой и экспонентой число 3000.0, например, будет выглядеть как 3*10^3, а 300.0 — как 3*10^2. (Здесь используется знак ^, хотя в C++ нет операции экспоненты.) При сложении чисел с плавающей точкой мантисса меньшего операнда сдвигается вправо, так что экспоненты двух операндов становятся одинаковыми (при сложении 3000 и 300 число 300 сдвигается на три десятичных позиции вправо и представляется как 3.3*10^3). После этого складываются биты мантиссы (а не все биты, как для целых). При сложении 3000 и 300 результатом будет 3.3*10^3.

Каковы бы ни были детали сложения чисел с плавающей точкой, ясно, что они отличаются от целочисленного сложения. На уровне языка ассемблера эти операции представлены разными кодами операций. Языки высокого уровня не вынуждают программистов заучивать разные обозначения для целочисленного сложения и сложения с плавающей точкой.

Остается надеяться, что вы нашли в данном обсуждении принципы сокрытия информации и переноса обязанностей с клиента на сервер. В таком случае, сервер — это операция сложения, а клиент — программный код высокого уровня, содержащий выражение с операцией сложения. Программист, использующий операцию сложения, не хочет знать детали операции. Он сосредотачивается на задачах более высокого уровня, на других связанных с ними вопросах и просто применяет операцию сложения, которая знает о деталях сложения каждого типа и реализуют соответственно каждую конкретную операцию.

C++ использует эти идею обозначения разных операций одним символом и выводит ее на новый уровень, расширяя такие возможности на типы данных, определяемые программистом. Если программист пишет функции в соответствии с правилами C++, то он может применять любую операцию (за некоторыми исключениями) к любому определяемому программистом типу!

Здесь операторная функция `operator+()` реализована для параметров типа `Complex`.

```
Complex operator+(const Complex &a, const Complex &b)       // волшебное имя
{ Complex c;                                  // локальный объект
  c.real = a.real + b.real;                   // сложение вещественных компонентов
  c.imag = a. imag + b.imag;                  // сложение мнимых компонентов
  return c; }
```

Как написана эта функция? Просто скопирована приведенная выше функция `addComplex()`, сохранено тело функции и список параметров, удалено имя функции `addComplex`, замененное на имя `operator+`. Дело сделано! C++ сделает и остальное. Он будет воспринимать операцию сложения с операндами типа `Complex` и не выведет сообщение о синтаксической ошибке, в которой говорится, что операция сложения для типа `Complex` не определена. Теперь эта операция определена.

```
Complex x, y, z;                              // объекты типа Complex
x.real = 20; x.imag = 40;                     // инициализация
y.real = 30; y.imag = 50;
z = x + y;                                    // использование в выражении
```

Что фактически означает фраза "компилятор воспринимает операцию сложения с операндами типа `Complex`"? Какой код он будет генерировать? Компилятор вызовет только что написанную перегруженную функцию `operator+()` и использует

левый операнд выражения как первый фактический аргумент функции, а правый как второй аргумент. Генерируемый компилятором код будет в точности совпадать со следующим клиентским кодом:

```
Complex x, y, z;                    // объекты типа Complex
x.real = 20; x.imag = 40;           // инициализация
y.real = 30; y.imag = 50;
z = operator+(x,y);                 // это абсолютно законно
```

Если имя функции включает в себя ключевое слово operator и символ операции, то компилятор воспринимает или вызов функции, или синтаксис операции и генерирует один тот же код. Когда используется синтаксис z = operator + (x,y);, он сопоставляет типы фактических аргументов с типами параметров (как для любого вызова функции). Если же применяется синтаксис z = x + y;, то компилятор понимает, что используются операнды определяемого программистом типа, и ищет функцию, у которой имя содержит ключевое слово operator и указанный в клиенте знак операции. Если эта функция найдена, то он проверяет соответствие ее параметров числу и типам операндов в клиентском выражении.

В результате обязанности переносятся на серверные классы, а клиентский код избавляется от деталей реализации серверов. Программист, работающий с клиентом, использует один и тот же синтаксис для сложения целых чисел, чисел с плавающей точкой, объектов типа Complex или каких-либо других объектов определяемого типа. И все — с помощью одного синтаксиса.

Это очень гибкий и мощный механизм. Как и многое другое в C++, программист получает больше, чем может ожидать. Мы начали с такой цели, как использование объектов определяемых программистом типов как переменных встроенных типов, а закончим более мощными средствами. Теперь можно делать с объектами собственных типов то, о чем и мечтать не приходилось при работе со встроенными числовыми типами. Ведь язык не ограничивает программиста в реализации перегруженных операторных функций. Ограничивается только интерфейс функции — ее имя и число параметров. Их нельзя выбирать произвольно. Они должны эмулировать перегруженные встроенные операции.

Перегрузка операторной функции проиллюстрирована в листинге 10.2. Кроме перегруженной операции сложения он показывает применение функции operator+=(), которая складывает один объект Complex с другим. Хотя имена этих двух операторных функций одинаковы, их список параметров различен. Это законное использование перегрузки имен (о совмещении имен функции см. в главе 7).

Листинг 10.2. Пример перегрузки операторных функций

```
#include <iostream>
using namespace std;

struct Complex {                                    // тип, определяемый программистом
  double real, imag; } ;

Complex operator+(const Complex &a, const Complex &b)  // волшебное имя
{ Complex c;                                        // локальный объект
  c.real = a.real + b.real;                         // сложение вещественных компонентов
  c.imag = a. imag + b.imag;                        // сложение мнимых компонентов
  return c; }

void operator += (Complex &a, const Complex &b)   // еще одно волшебное имя
{ a.real = a.real + b.real;                       // сложение вещественных компонентов
  a.imag = a. imag + b.imag;}                     // сложение мнимых компонентов

void operator += (Complex &a, double b)           // другой интерфейс
{ a.real += b; }                                  // сложение только вещественных компонентов
```

```
void showComplex(const Complex &x)
{ cout << "(" << x.real << ", " << x.imag << ")" << endl; }

int main()
{  Complex x, y, z1, z2;                          // объекты типа Complex
   x.real = 20; x.imag = 40;                      // инициализация
   y.real = 30; y.imag = 50;
   cout << "Первое значение: "; showComplex(x);
   cout << "Второе значение: "; showComplex(y);
   z1 = operator+(x,y);                           // использование в вызове функции
   cout << "Сумма как вызов функции: "; showComplex(z1);
   z2 = x + y;                                    // использование как операции
   cout << "Сумма как операция: "; showComplex(z1);
   z1 += x;                                       // то же, что operator+=(z1,x);
   cout << "Сложение первого значения с суммой: "; showComplex(z1);
   z2 += 30.0;                                    // то же, что operator+=(z2,30.0);
   cout << "Сложение 30 с суммой: "; showComplex(z2);
   return 0;
}
```

Обратите внимание на использование там, где это требуется, ключевого слова const: в showComplex(), в operator+() и в operator+=(), а также на его отсутствие в первой функции operator+=() и второй operator+=(). Отметьте в данном примере некоторые преимущества объектно-ориентированного программирования. Клиент не зависит от архитектуры сервера и имен полей данных (не требующих инициализации), а вычисления нижнего уровня перенесены в серверные функции. Смысл вычислений высокого уровня выражается с помощью вызовов функций. Вычисления нижнего и высокого уровня решают разные вопросы. На нижнем уровне обрабатываются поля комплексных чисел (согласно правилам комплексной арифметики), на верхнем — операции с комплексными числами выполняются согласно стоящим перед приложением целям. Соответственно для представления данных и для алгоритма приложения области изменения различны. Если меняется архитектура класса Complex, то придется модифицировать только перегруженные операции, а не клиент. При изменении алгоритма приложения модифицируется клиент, но не перегруженные операции.

В то же время здесь отсутствуют некоторые преимущества объектно-ориентированного программирования: инкапсуляция "добровольна", ничто не указывает на общую принадлежность данных и серверных функций, а имена функций глобальны. Звучит знакомо? Хорошо. Вот мы и добрались до цели.

Результат данной программы представлен на рис. 10.2.

Как видно из листинга 10.2, ключевое слово operator можно отделять (или не отделять) от знака операции пробелами. Допускается запись operator+() и operator + (). Если знак операции содержит два символа, то эти символы не должны разделяться пробелами.

```
Первое значение:  <20, 40>
Второе значение:  <30, 50>
Сумма как вызов функции:              <50, 90>
Сумма как операция:                   <50, 90>
Сложение первого значения с суммой: <70, 130>
Сложение 30 с суммой:                 <80, 90>
```

Рис. 10.2. *Результат выполнения программы из листинга 10.2*



Внимание Обязательными компонентами имени перегруженной операторной функции являются ключевое слово operator и символ (или символы) операции. Они составляют имя функции и могут разделяться пробелами (их можно включить в имя, если это облегчает чтение программы). Такое разделение не считается синтаксической ошибкой.

Нужно иметь в виду, что в случае использования перегруженных операций в пространстве клиента, они реализуются как вызовы функций. Применение перегруженных операций не ускорит выполнение программы. Это лишь синтаксический прием, позволяющий сделать ее более удобной для чтения. Во всех случаях вместо синтаксиса перегруженных операций допустимо использовать вызовы функций. Например, последнюю часть клиентской программы из листинга 10.2 можно записать так:

```
operator+=(z1,y);                                    // то же, что z1 += x;
cout << "Сложение первого значения с суммой: " ; showComplex(z1);
operator+=(z2,30.0);                                 // то же, что z2 += 30.0;
cout << "Сложение 30 с суммой:                 "; showComplex(z2);
```

Конечно, нет смысла создавать перегруженные операторные функции только для того, чтобы использовать их в клиенте как вызов функции. Для этого можно вызвать функцию `addComplex()`, а не `operator+()`. Перегруженные операторные функции применяются для того, чтобы воспользоваться возможностями компилятора C++, который сам интерпретирует их вызов как обычный вызов функции. При этом нужно не забывать о том, что они компилируются в вызов функции, а не в арифметические выражения, как подчас кажется.

Ограничения перегрузки операций

Как показано в предыдущем разделе, перегруженные операции — это мощный инструмент, позволяющий интерпретировать в C++ определяемые программистом объекты подобно переменным встроенных типов и делающий программы более привлекательными. Но есть некоторые ограничения, налагаемые на использование перегруженных операций. Часть из них не особенно влияет на работу программиста, другие — весьма существенны. О них рассказывается в данном разделе.

Какие операции не могут быть перегруженными

Некоторые ограничения на перегрузку операций не имеют для практики программирования особо важного значения. Не допускается перегрузка операций `::` (область действия), `.*` (выбор компонента объекта), `.` (выбор объекта класса) и `?:` (условный оператор или арифметическое ИЛИ). Трудно представить, для чего кому-то может понадобиться совмещать операцию области действия или условия, как и операцию выбора компонента объекта и объекта класса. (На самом деле операцию выбора компонента объекта мы не используем здесь даже в своем первоначальном смысле.)

С практической точки зрения важно следующее ограничение: нельзя создавать собственные операции, не поддерживаемые для числовых типов C++. Операция, знак которой добавляется к ключевому слову `operator` в перегруженной операторной функции, должна быть допустимой в C++. Применение символа, не являющегося операцией C++, будет синтаксической ошибкой. Например, в C++ нет операции возведения в степень. В других языках она существует и обозначается как `**`. Например, в Фортране `x**y` означает x в степени y. Возможно, кому-то захочется расширить набор операций C++ двойной звездочкой:

```
Complex operator**(const Complex &a, const Complex &b);          // ошибка
```

Это ошибка, так как в C++ нет встроенной операции `**`.

Перегруженные операции нельзя применять для встроенных числовых типов, придавая им новый смысл. Например, в приложении может потребоваться ограничить результаты целочисленного сложения конкретным числом, допустим 60

(арифметика по модулю), для чего делается попытка переопределить операцию сложения целых чисел, установив данное ограничение:

```
int operator + (int a, int b)          // синтаксическая ошибка
{ return (a+b) % 60)}                  // сложение по модулю 60
```

Хорошая идея, но она не сработает по нескольким причинам. Основная причина в том, что при перегрузке нескольких операций компилятор запутается в их разных дополнительных смыслах. Так, в перегруженной операции для целых чисел в последнем примере в теле функции используется операция сложения. Хотелось бы применять ее здесь в стандартном смысле, но как сообщить об этом компилятору? Откуда он узнает, что это не рекурсивный вызов функции operator+()?

Такие же трудности возникают в коде клиента.

```
int a,b,c;
float d,e,f;
a = 20, b = 30; d = 40.0; e = 50.0;

// встроенная или перегруженная операция?
c = a + b; f = d + e;
```

Здесь никак нельзя сообщить компилятору, что нужно использовать — встроенную операцию сложения или перегруженную операцию.

Именно поэтому C++ не допускает перегрузки операций для встроенных типов. Можно перегружать операции только для типов, определяемых программистом. Фактически, в C++ это ограничение обобщается: требуется, чтобы как минимум один параметр в перегруженной операторной функции был параметром определяемого программистом типа (классом). В показанной выше попытке ввести дополнительную операцию для сложения целых чисел это ограничение нарушается.



Осторожно! Набор операций C++ нельзя расширить за счет символов перегруженных операций, не используемых в качестве встроенных операций C++. Можно перегружать только существующие операции C++. Нельзя также менять смысл операций со встроенными типами. Разрешается перегружать только операции для типов, определяемых программистом (классов).

Обратите внимание, что во всех случаях создаются перегруженные операторные функции, а не переопределяются существующие операции. Добавление операции для объектов Complex не исключает операции сложения для целых чисел и чисел с плавающей точкой. Перегруженная операция просто добавляется к списку операций, известных компилятору C++. Давайте снова рассмотрим эту перегруженную операцию:

```
Complex operator+(const Complex &a, const Complex &b)       // волшебное имя
{ Complex c;                                  // локальный объект
  c.real = a.real + b.real;                   // сложение вещественных компонентов
  c.imag = a.imag + b.imag;                   // сложение мнимых компонентов
  return c; }
```

Операция сложения в теле такой перегруженной операторной функции представляет собой стандартную встроенную операцию для чисел с плавающей точкой. Откуда это известно компилятору? По типам полей данных класса Complex. Поскольку поля имеют тип double, операция сложения не является здесь рекурсивным вызовом определяемой перегруженной операции. Аналогичный анализ применяется и к клиентскому коду:

```
Complex x, y, z;                  // объекты типа Complex
x.real=20; x.imag=40;             // инициализация
```

```
y.real=30; y.imag=50;
z = x + y;                    // использование в выражении
double a, b, c;               // переменные типа double
a = 20; b = 30;               // инициализация
c = a + b;                    // использование в выражении
```

Встречая первую операцию сложения, компилятор устанавливает, что операнды имеют тип Complex, и вызывает перегруженную операторную функцию. Для второй операции сложения он вызывает встроенную операцию, так как операнды имеют тип double.

Ограничения на возвращаемые типы

Обычно перегруженные операторные функции возвращают либо void, либо булево значение, либо значение того типа, для которого предназначена операция. Часто возвращаются значения типа класса. Это особенно популярно в случае операций, вычисляющих для применения в других выражениях новое значение того же типа. Например, operator+() в приведенных выше примерах возвращает значение типа Complex, что позволяет использовать данное значение с операцией присваивания. Если бы возвращалось значение типа void, то такое было бы невозможно. Кроме того, возврат значения открывает путь к построению более сложных выражений, аналогичных выражениям со встроенными типами.

```
Complex a, b, c, d;           // объекты типа Complex
a.real=20; a.imag=40;         // инициализация
b.real=30; b.imag=50;
c.real=0; c.imag=20;
d = a + b + c;                // использование в выражении
```

Требуется некоторое время, чтобы уяснить суть этого программного кода. Арифметические выражения C++ ассоциируются слева направо. Если бы a, b и c были числами, то выражение a + b + c означало бы (a + b) + c. Перегруженные операторные функции не меняют ассоциативности операций. Если a, b и c — объекты типа Complex, то смысл выражения будет тем же:

```
d = (a + b) + c;              // использование в выражении
```

В терминах синтаксиса функций выражение означает следующее:

```
d = operator+((a + b),c);     // использование в выражении
```

Осталось представить выражение a + b как вызов функции:

```
d = operator+(operator+(a,b),c);   // использование в выражении
```

Смысл данного кода в том, что здесь вызывается функция operator+() с переменными a и b, передаваемыми в фактических аргументах, а возвращаемое функцией значение становится первым аргументом функции при втором вызове operator+().

Две перегруженные операторные функции operator+=() из листинга 10.2 возвращают тип void, следовательно, не могут применяться в цепочке, где для дальнейших операций в выражении используется полученное значение.

```
Complex a, b, c, d;           // объекты типа Complex
a.real=20; a.imag=40;         // инициализация
b.real=30; b.imag=50;
c.real = 0; c.imag = 20;
d = a + b + c;                // использование в выражении
a += b;                       // OK, operator+=(a,b); возвращает void
d = c + (b += 30.0);          // нехорошо: operator+=(b,30.0); возвращает void
```

Чтобы эту операцию можно было использовать в цепочке выражений, нужно построить программу так:

```
Complex operator += (Complex &a, double b)          // класс возвращает тип
{ a.real += b;                                      // сложение с real
  return a; }
```

Таким образом можно лучше моделировать поведение встроенных типов, хотя поведение встроенных типов C++ имеет свои недостатки. Оно способствует не разбиению алгоритма на простые последовательные шаги, а написанию запутанных выражений. Вместо изменения программного кода сервера (перегруженной операторной функции) для организации цепочки операций в программе лучше было бы возвращать тип void и разбить код клиента на мелкие шаги, не обязательно возвращающие значение типа Complex.

```
Complex a, b, c, d;           // объекты типа Complex
a.real=20; a.imag=40;         // инициализация
b.real=30; b.imag=50;
c.real = 0; c.imag = 20;
d = a + b + c;      // использование в выражении
a += b;             // OK, operator+=(a,b); возвращает void
b += 30.0;          // OK, operator+=(b,30,0); возвращаемый тип не используется
d = c + b           // OK: operator+=(c,b); возвращает Complex
```

Здесь все дело вкуса. Но нужно видеть и понимать разные способы организации взаимодействия сервера и клиента.

Ограничения на число параметров

При проектировании перегруженных операторных функций следует использовать столько аргументов (обычно того же типа класса), сколько необходимо для операции (бинарной или унарной).

Изменять арность операции нельзя, т. е. нужно точно задавать число операндов, которые будут с нею использоваться (два для бинарной операции, один — для унарной). Арность перегруженной операции должна быть такой же, как у оригинальной встроенной операции. Нельзя определить бинарную операцию, работающую с двумя операндами, и использовать ее для создания унарной операции с одним операндом.

Стоит привести пример типичных трудностей, с которыми приходится сталкиваться в борьбе с данными правилами. Предположим, нужно реализовать перегруженную операцию "меньше чем" (<) для элементов данных типа Complex и организации вывода данных, выполняемого функцией showComplex() из листинга 10.2.

Для этого требуется лишь заменить имя showComplex на operator <.

```
void operator < (const Complex &x)      // плохая идея: синтаксическая ошибка
{ cout << "(" << x.real << ", " << x.imag << ")" << endl; }
```

Нетрудно увидеть, как можно использовать такую функцию в клиенте. Она вызывается так же, как функция showComplex():

```
Complex x, y, z1, z2;                   // объекты типа Complex
x.real = 20; x.imag = 40;               // инициализация
y.real = 30; y.imag = 50;
cout << "Первое значение: ";
operator < (x);                         // то же, что showComplex();
cout << "Второе значение: ";
operator < (y);                         // то же, что showComplex();
```

Между тем операция "меньше чем" — двухместная операция (операция с двумя операндами), и использование этой функции с синтаксисом операции требует второго операнда, который пропущен.

```
Complex x, y, z1, z2;                    // объекты типа Complex
x.real = 20; x.imag = 40;                // инициализация
y.real = 30; y.imag = 50;
cout << "Первое значение: ";
< x;                                     // нонсенс, если x - числовое значение
cout << "Второе значение: ";
< y;                                     // нонсенс, если y - числовое значение
```

Итак, первая попытка не удалась. Операторная функция `operator < ()` должна иметь два параметра, а здесь только один — выводимый объект типа `Complex`. Если неизвестно, что делать с другим операндом, пришлось бы найти другую, "одноместную" операцию.

В C++ есть несколько операций, которые можно использовать как бинарные и унарные: плюс, минус, звездочка и знак @. Можно перегружать каждую из них как операцию с одним или двумя операндами. Ведь для встроенных типов возможны оба варианта. Например, в нашем примере операция + перегружается как бинарная. Поскольку она доступна и как унарный плюс, можно совместить ее, используя функцию с одним параметром. Такая замена для `showComplex()` вполне законна:

```
void operator + (const Complex &x)              // то же, что и showComplex()
{ cout << "(" << x.real << ", " << x.imag << ")" << endl; }
```

Можно без проблем использовать эту функцию, применяя в клиенте синтаксис операции:

```
Complex x, y, z1, z2;          // объекты типа Complex
x.real = 20; x.imag = 40;      // инициализация
y.real = 30; y.imag = 50;
cout << "Первое значение: ";
+ x;                           // operator+(x); или showComplex(x);
cout << "Второе значение: ";
+ y;                          // то же, что operator+(y); или showComplex(y);
```

Ограничение на старшинство операций

При перегрузке операций действует еще одно ограничение. Нельзя изменять старшинство перегруженных операций.

Независимо от типа объектов x и y и смысла операций + и /, деление в выражении x + y/2 будет выполняться перед сложением. Если нужен другой порядок, следует, как обычно, использовать скобки.



Осторожно! При перегрузке операторных функций нельзя изменить число операндов в операции, старшинство операций или их ассоциативность. Разрешается лишь изменять смысл операции для определяемых программистом типов. Это позволяет клиенту применять для определяемых программистом классов и стандартных встроенных типов один и тот же синтаксис выражений.

Перегруженные операции как компоненты класса

Как уже упоминалось в предыдущей главе, любую функцию, связанную с любым определенным программистом типом данных, можно реализовать как функцию-член класса или как глобальную автономную функцию, не являющуюся элементом

класса. Это применимо к любому алгоритму и к перегруженным операторным функциям. Переход от компонентов класса к некомпонентной реализации и обратно — очень важный для программиста навык, особенно для операторных функций.

Операцию можно определить как функцию-член класса с параметрами, число которых на единицу меньше арности операции (1 для бинарной, 0 для унарной). При использовании операции отсутствующий параметр становится адресатом сообщения.

Замена глобальной функции компонентом класса

Правила перегрузки операторных функций-членов класса совпадают с правилами перегрузки операторных функций как некомпонентных функций. Имя функции-члена заменяется именем со словом `operator` и знаком определяемой операции.

Например, бинарные операции `operator+()` и `operator+=()`, реализованные как функции-члены класса `Complex`, должны иметь только один параметр, а не два, как операции в листинге 10.2, реализованные через глобальные функции. Элементы данных параметра, "исчезнувшего" из интерфейса функции, стали элементами данных адресата сообщения.

```
class Complex {                          // тип, определенный программистом
    double real, imag;                   // закрытые данные
public:
    Complex(double r, double i)          // общий конструктор
    { real =r; imag = i; }

Complex operator+(const Complex &b)      // только один параметр
{ Complex c;                             // работает?
  c.real = real + b.real;                // сложение вещественных компонентов
  c.imag = imag + b.imag;                // сложение мнимых компонентов
  return c; }

void operator += (const Complex &b)      // только один параметр
{ real = real + b.real;                  // сложение вещественных компонентов
  imag = imag + b.imag;                  // сложение мнимых компонентов
                                         // остальная часть класса Complex
} ;
```

Освоившись с этим, будете комфортно чувствовать себя при переходе от некомпонентной функции с двумя параметрами к функции-члену с одним параметром. Многие программисты предпочитают реализацию с двумя параметрами, поскольку она симметрична: можно складывать соответствующие поля двух параметров.

```
Complex operator+(const Complex &a, const Complex &b)
                                                    // только один параметр
{ Complex c; // работает?

  // сложение компонентов: симметричная запись
  c.real = a.real + b.real;
  c.imag = a.imag + b.imag;
  return c; }

void operator += (Complex &a, const Complex &b)        // глобальная функция
{ a.real = a.real + b.real;              // сложение вещественных компонентов
  a.imag = a.imag + b.imag; }            // сложение мнимых компонентов
```

При преобразовании первой операторной функции в функцию-член возникает одна проблема. Здесь используется локальная переменная типа `Complex`, которая не инициализируется, поскольку неважно, какие значения имеют ее элементы

данных. Эти значения все равно перезаписываются функцией до возврата результата клиенту. Такая же схема использовалась в листинге 10.2, и там она работала. Для класса Complex не предусматривались конструкторы — система просто использовала конструктор по умолчанию (не выполняющий операций). В новом варианте, приведенном в данном разделе, класс Complex имеет общий конструктор, и, следовательно, компилятор использует его и сообщает о попытке вызова несуществующей функции в первой строке операторной функции. Таким образом, никогда не нужно забывать о конструкторах.

Преодолеть проблему можно двумя способами. Первый состоит в инициализации локального объекта какими-то ненужными значениями.

```
Complex operator+(const Complex &b)       // только один параметр
{ Complex c(0,0);                         // попытка умиротворить компилятор
  c.real = real + b.real;    // сложение компонентов: несимметричная реализация
  c.imag = imag + b.imag;
  return c; }
```

Лучший же способ состоит в том, чтобы исключить локальный объект. Вместо него создается неименованный объект Complex, инициализируемый результатами вычислений, и из операторной функции возвращается значение этого неименованного объекта:

```
Complex operator+(const Complex &b)                    // только один параметр
{ return Complex (real + b.real; imag + b.imag); }  // хорошо: быстро и изящно
```



Внимание Убедитесь, что архитектура класса поддерживает не только клиент, но и свои собственные методы. Отсутствие необходимых конструкторов — распространенный источник проблем при проектировании классов.

В клиенте для объектов Complex можно использовать либо синтаксис вызова функции, либо синтаксис операции. В приведенном ниже примере применяются оба варианта. Вызов функции-члена отличается от вызова функции, не являющейся членом класса: один из параметров становится получателем сообщения.

```
Complex x(20,40),y(30,60),z1(0,0),z2(0,0);              // созданные объекты
z1 = x.operator+(y);          // используется как сообщение, передаваемое x
z2 = x + y;                   // то же, что z2=x.operator+(y);
z1.operator+=(y);             // используется как сообщение, передаваемое z1
z2 += x;                      // то же, что z2.operator+=(x);
```

Синтаксис операции для функции-члена в точности такой же, как для не члена. Компилятор различает эти вызовы и может интерпретировать выражения, включающие в себя вызовы функций-методов с именами, которые содержат ключевое слово operator, и соответствующий знак встроенной операции:

```
z2 = x + y;                   // то же, что z2=x.operator+(y);
z2 += x;                      // то же, что z2.operator+=(x);
```

Для функций-членов синтаксис вызова функции можно использовать точно так же, как вызовы глобальных функций. К таким способам прибегают немногие программисты, и здесь они приводятся только для уяснения действительного смысла синтаксиса операций в выражениях.

```
z2=x.operator+(y);            // то же, что z2 = x + y;
z2.operator+=(x);             // то же, что z2 += y;
```

Что происходит при перегрузке для одной и той же операции и глобальной функции, и функции-члена? Такую идею хорошей не назовешь. Если эти функции вызываются с помощью синтаксиса вызова функций, то компилятор не поймет, что именно имеется в виду. Если же применяется синтаксис операций, то компилятор все равно запутается. Ведь подойдет и та, и другая функция. Обе имеют одинаковое старшинство, и компилятор отвергнет такое выражение как неоднозначное.

Использование членов класса для цепочек операций

Аналогично некомпонентной реализации возвращаемый функцией-членом тип void препятствует использованию синтаксиса операции в цепочке операций. При возврате объекта класса можно организовать выражение-цепочку:

```
Complex a(20,40), b(30,50), c(0,20), d(0,0);  // определение и инициализация
d = a + b + c;                                // использование в цепочке выражений
```

Вновь встроенная операция + ассоциируется слева. Перегруженная операция + также ассоциируется слева, и смысл цепочки операций будет таким:

```
d=(a + b) + c;
```

Синтаксис бинарной операции скрывает из виду то, что мы имеем дело с сообщением operator+(), передаваемым экземпляру объекта класса Complex. Смысл a + b для реализации функции-члена — это a.operator+(b). Следовательно, клиентское выражение d = (a + b) + c; в точности эквивалентно следующему:

```
d = a.operator+(b) + c;    // сообщение возвращаемому значению a.operator+(b)
```

Здесь operator+ также представляет сообщение, передаваемое объекту и возвращаемое первым вызовом функции. Следовательно, выражение-цепочка имеет такой смысл:

```
d=a.operator+(b).operator+(c);          // сообщение возвращаемому значению
```

Нужно обязательно освоить интерпретацию таких цепочек, как вызовов функций.

Переопределение операции имеет место только в контексте класса, где определена перегруженная операция. Оно используется, когда объекту типа Complex передается сообщение (с параметром типа Complex), поэтому в самом определении функции-члена в c.real = real + b.real; используется стандартный смысл символа +. Это не рекурсивный вызов перегруженной операции +. Во встроенной функции операция + применяется к значениям типа double. Компилятор знает, что левый операнд имеет тип double. Это не объект, и он не может быть получателем сообщения operator+(). Для значений double используется встроенная операция +.

В листинге 10.3 показана новая версия класса Complex и реализация других перегруженных операций как функций-членов. Вторая функция operator+=() теряет свой параметр Complex. Вместо него используется целевой объект Complex. Унарная функция operator+(), реализующая функцию showComplex(), теперь вовсе не имеет параметров. Это не противоречит правилу, что перегруженная операторная функция (реализуемая, как не член) должна иметь по крайней мере один объект-параметр типа класса. Данный объект и есть адресат сообщения. Для реализации операции вывода большинство программистов предпочли бы совмещать операцию <<, а не операцию +. Ниже будет показано, как это сделать. Результат данной программы представлен на рис. 10.3.

```
Значение x:    <20, 40>
Значение y:    <30, 50>
z1 = x + y:        <50, 90>
z2 = x + y:        <50, 90>
Сложение x с z1:   <70, 130>
Сложение 30 с z2: <80, 90>
```

Рис. 10.3.
Результат выполнения программы из листинга 10.3

Листинг 10.3. Перегруженные операторные функции как члены класса

```
#include <iostream>
using namespace std;

class Complex {                                  // тип, определяемый программистом
   double real, imag;                            // закрытые данные
public:                                          // общедоступные функции-члены

Complex(double r, double i)                      // общий конструктор
   { real =r; imag = i; }

Complex operator+(const Complex &b)              // только один параметр
{ return Complex (real = b.real. imag + b.imag); }          // быстро и изящно

void operator += (const Complex &b)              // изменяется ли целевой объект?
{  real = real + b.real;                         // сложение вещественных компонентов
   imag = imag + b.imag; }                       // сложение мнимых компонентов

void operator += (double b)                      // другой интерфейс
{ real += b; }

void operator + ()                               // используется как showComplex(const Complex &x)
{ cout << "(" << real << ", " << imag << ")" << endl; }
};                                               // конец класса Complex

int main()
{  Complex x(20,40), y(30,50), z1(0,0), z2(0,0);            // созданные объекты
   cout << "Значение x: "; +x;                   // то же, что x.operator+();
   cout << "Значение y: "; y.operator+();        // и так можно
   z1 = x.operator+(y);                          // использование в вызове функции
   cout << "z1 = x + y: ";
   +z1;                                          // вывод z1
   z2 = y + y;                                   // то же, что z2=x.operator+(y);
   cout << "z2 = x + y: ";                       // использование как операции
   +z2;                                          // вывод z2
   z1 += x;                                      // то же, что z1.operator+=(x);
   cout << "Сложение x с z1: "; +z1;
   z2 += 30.0;                                   // то же, что z2.operator+=(30.0);
   cout << "Сложение 30 с z2: "; +z2;
   return 0;
   }
```

Применение ключевого слова const

Использование ключевого слова const с параметрами функции здесь ничем не отличается от листинга 10.2. Нет никаких причин что-либо менять. Между тем некоторые параметры из листинга 10.2 в листинге 10.3 отсутствуют. Вот как выглядят глобальные функции-серверы из листинга 10.2:

```
Complex operator+(const Complex &a, const Complex &b)       // волшебное имя
{ Complex c;                                 // локальный объект
  c.real = a.real + b.real;                  // сложение вещественных компонентов
  c.imag = a. imag + b.imag;                 // сложение мнимых компонентов
  return c; }

void operator += (Complex &a, const Complex &b)     // еще одно волшебное имя
{ a.real = a.real + b.real;                  // сложение вещественных компонентов
  a.imag = a. imag + b.imag;}                // сложение мнимых компонентов
```

```
void operator += (Complex &a, double &b)                    // другой интерфейс
{ a.real += b; }                                  // сложение вещественных компонентов

void showComplex(const Complex &x)      // это operator+() в листинге 10.3
{ cout << "(" << x.real << ", " << x.imag << ")" << endl; }
```

В листинге 10.2 разработчик выразил свое знание о первом параметре функции operator+() с помощью ключевого слова const. Аналогично первые параметры обеих функций operator+=() выражают это знание в форме отсутствия ключевого слова const. В листинге 10.3 данных параметров нет. Как можно отразить отсутствие или наличие ключевого слова const для таких объектов? Ведь исчезли они лишь из интерфейса функции, но не из приложения, что особенно ясно из синтаксиса операции в клиенте.

```
Complex x(20,40), y(30,50), z1(0,0), z2(0,0);  // определение, инициализация
z2 = x + y;                                    // x и y здесь не изменяются
z1 += x;                                       // z1 изменяется в результате операций
z2 += 30.0;                                    // z2 изменяется в результате операций
```

Как бы ни реализовывались эти операции — как автономные функции или как функции-члены, операнды в правой части не изменяются. В результате операции изменяются операнды в левой части. Из анализа синтаксиса вызова функции это не следует. (Не забывайте, что синтаксис оператора — просто альтернативная форма, допустимая при следовании ограничениям языка.)

```
Complex x(20,40), y(30,50), z1(0,0), z2(0,0);  // определение, инициализация
z2 = x.operator + (y);                         // x при вызове не изменяется
z1.operator += (x);                            // z1 изменяется в результате операций
z2.operator += (30.0);                         // z2 изменяется в результате операций
```

Итак, как же показать, что элементы данных объекта (адресат сообщения) при выполнении метода не изменяются? С помощью ключевого слова const. Но где оно должно находиться? Между закрывающей круглой скобкой списка параметров и открывающей фигурной скобкой тела функции. В прототипе функции оно включается между закрывающей скобкой списка параметров и точкой с запятой. Мы уже говорили, что следует всегда думать об использовании ключевого слова const. Так оно и есть.

В листинге 10.4 приведена та же программа, что в листинге 10.3, но, там где нужно, добавлено ключевое слово const. Кроме того, комплексные функции-члены реализованы вне границ класса (фигурных скобок). При этом пришлось использовать операцию области действия класса. Она применяется к перегруженным операторным функциям точно так же, как к любым другим функциям-членам. Конечно, ключевое слово const повторяется в прототипе функции и в ее реализации. Любое различие помечается как синтаксическая ошибка (возможно, в сообщении будет говориться совсем о другом). Результат программы соответствует рис.10.3.

Листинг 10.4. Перегруженные операторные функции, реализованные вне спецификации класса

```
#include <iostream>
using namespace std;

class Complex {                                    // тип, определяемый программистом
  double real, imag;                               // закрытые данные
public:                                            // общедоступные функции-члены
  Complex(double r, double i)                      // общий конструктор
  Complex operator+(const Complex &b) const        // целевой объект не изменяется
  void operator += (const Complex &b)              // целевой объект изменяется
```

```
   void operator += (double b)                          // целевой объект изменяется
   void operator + () const                             // целевой объект не изменяется
};                                                      // конец класса Complex

Complex::Complex(double r, double i)                    // общий конструктор
   { real =r; imag = i; }

Complex::Complex operator+(const Complex &b) const
{ return Complex (real = b.real, imag + b.real); }

void Complex::operator += (const Complex &b)            // изменяется целевой объект
{ real = real + b.real;                 // сложение с вещественными компонентами целевого объекта
   imag = imag + b.imag; }              // сложение с мнимыми компонентами целевого объекта

void Complex::operator += (double b)                    // целевой объект изменяется
{ real += b; }                          // сложение вещественного компонента с целевым объектом

void Complex::operator + () const                       // целевой объект не изменяется
{ cout << "(" << real << ", " << imag << ")" << endl; }

int main()
{  Complex x(20,40), y(30,50), z1(0,0), z2(0,0);        // определение, инициализация
   cout << "Значение x: "; +x;                          // то же, что x.operator+();
   cout << "Значение y: "; y.operator+();               // и так можно
   z1 = x.operator+(y);                                // использование в вызове функции
   cout << "z1 = x + y: ";+z1;
   z2 = y + y;                                         // то же, что z2=x.operator+(y);
   cout << "z2 = x + y; "; +z2
   z1 += x;                                            // то же, что z1.operator+(y);
   cout << "Сложение x с z1: "; +z1;
   z2 += 30.0;                                         // то же, что z2.operator+=(30.0);
   cout << "Сложение 30 с z2: "; +z2;
   return 0;
   }
```



Советуем Используйте ключевое слово const для параметров перегруженных операторных функций, не изменяющих значений этих параметров. При реализации перегруженных операций как функций-членов не забывайте применять ключевое слово const к целевому объекту. Если целевой объект не изменяется, функция должна быть помечена как const. Отсутствие ключевого слова const должно говорить о том, что целевой объект изменяется.

Учебный пример: рациональные числа

В данном разделе обсуждается еще один популярный пример перегруженных операторных функций — класс, инкапсулирующий реализацию рациональных чисел (точных дробей). Он реализует арифметические операции и сравнения, не поддерживаемые для целых чисел.

Рациональные числа можно представить в виде двух компонентов: числителя и знаменателя. Они позволяют выполнять операции с дробями без ошибок округления, например 1/4 + 3/2 = 14/8 = 7/4.

В реализации класса числитель и знаменатель можно сделать закрытыми элементами данных. Если приложение планируется переносить на 16-разрядную машину, то элементы данных следует определить как long. Если же оно будет выполняться только на 32-разрядной машине, то элементы данных могут иметь тип int или long (в этом случае они представляют одинаковый диапазон).

```
class Rational {
    long nmr;
    long dnm;                    // закрытые данные
public:
    Rational()                   // конструктор по умолчанию: нулевые значения
  { nmr = 0; dnm = 0; }          // плохая идея
    Rational(long n, long d)     // общий конструктор: дробь как n/d
  { nmr = n; dnm = d; }
                                 // ОСТАЛЬНАЯ ЧАСТЬ КЛАССА Rational
    } ;
```

Общий конструктор инициализирует поля объекта значениями, заданными клиентом. Конструктор по умолчанию может создавать для дальнейшего присваивания значений неинициализированные объекты. Большинство программистов не любят оставлять поля объекта неинициализированными, поэтому используют те или иные значения по умолчанию. В данном примере объект инициализируется нулевым значением.

```
Rational a(1,4), b(3,2), c, d;
c = a + b;                              // 1/4+3/2 = (1*2+4*3)/(4*2)=14/8=7/4;
                                        // c.nmr есть 7, c.dnm есть 4
```

Может ли конструктор по умолчанию инициализировать оба элемента данных нулем? Если объект используется не как r-значение, а только как l-значение, то подобно объекту c в приведенном примере, никакого вреда это не принесет. Однако если работающий с клиентом программист предполагает, что неинициализированный объект всегда инициализируется нулем, и использует такой объект в вычислениях (например для подсчета суммы), то могут возникнуть проблемы.

```
Rational a(1,4), b(3,2), c, d;
c = a + b;                    // c.nmr есть 7, c.dnm есть 4
d += b;                       // 0/0 + 3/2 = (0*2+3*0)/(0*2); d.nmr=0, d.dnm=0
```

Вот почему лучше присваивать знаменателю ненулевое значение, например 1:

```
Rational::Rational()
{ nmr = 0; dnm = 1; }        // нулевое значение в виде 0/1
```

Благодаря этому конструктору по умолчанию объекты `Rational` можно использовать как r-значения и как l-значения. Когда он используется как l-значение (объект "c" ниже), вызов конструктора работает "вхолостую".

```
Rational a(1,4), b(3,2), c, d;
c = a + b;                    // c.nmr есть 7, c.dnm есть 4
d += b;                       // 0/1 + 3/2 = (0*2+3*0)/(0*2); d.nmr=3, d.dnm=2
```

Арифметические операции могут реализовываться как перегруженные операторные функции, которые следуют правилам операций с дробями. Ниже приведена операторная функция `operator+()`, поддерживающая сложение двух объектов `Rational`. Компонент на первой строке функции описывает следующий алгоритм: числитель результата есть сумма произведений числителя одной дроби на знаменатель другой, а знаменатель — результат произведения знаменателей операндов.

```
Rational Rational::operator + (const Rational &x) const
{  Rational temp;                  // n1/d1+n2/d2 = ((n1*d2)+(n2*d1))/(d1*d2)
   temp.nmr = (nmr * x.dnm) + (x.nmr * dnm);
   temp.dnm = dnm * x.dnm;         // например, 1/4 + 3/2 = 14/8
   return temp; }
```

Проблема данной реализации в том, что она не нормализует результат. Во-первых, это неудобно для пользователя. Во-вторых, знаменатели при вычислениях увеличиваются, а значит, возможно переполнение. Чтобы избежать этого, класс Rational должен поддерживать алгоритм нормализации, вызываемый в конце каждой арифметической операции (включая создание объекта).

```
class Rational {
    long nmr; dnm;              // закрытые данные
public:
    Rational()                  // конструктор по умолчанию: нулевые значения
  { nmr = 0; dnm = 1; }
    Rational(long n, long d)   // общий конструктор: дробь как n/d
  { nmr = n; dnm = d;
    normalize(); }
Rational operator + (const Rational &x) const        // важное ключевое слово
{ Rational temp;               // n1/d1+n2/d2 = ((n1*d2)+(n2*d1))/(d1*d2)
  temp.nmr = (nmr * x.dnm) + (x.nmr * dnm);
  temp.dnm = dnm * x.dnm;      // например, 1/4 + 3/2 = 14/8
  temp.normalize();
  return temp; }

void normalize()               // найти наибольший общий делитель
{ if (nmr == 0) { dnm = 1; return; }          // если ноль, ничего не делается
  int sign = 1;                // если число отрицательное, превратить в -1
  if (nmr < 0) { sign = -1; nmr = -nmr; } // сделать оба числа положительными
  if (dnm < 0) { sign = -sign; dnm = -dnm; }
  long gcd = nmr, value = dnm;           // поиск наибольшего общего делителя
  while (value != gcd) {  // остановить, когда найден наименьший общий делитель
    if (gcd > value)
      gcd = gcd - value;                  // вычесть меньшее из большего
    else value = value - gcd; }
  nmr = sign * (nmr/gcd); dnm = dnm/gcd; }          // делитель положительный
                                         // ОСТАЛЬНАЯ ЧАСТЬ КЛАССА Rational
  } ;
```

Можно проанализировать математическую часть алгоритма нормализации. Но нас интересует программирование, а не математика, и потому займемся вопросами программирования.

	nmr	dnm	gcd	value	value != gcd	gcd > value
Начало алгоритма	14	8			<-(начальные значения полей)	
Перед циклом			14	8	true: итерация	true: уменьшение gcd
После первого прохода			6	8	true: итерация	false: уменьшение value
После второго прохода			6	2	true: итерация	true: уменьшение gcd
После третьего прохода			4	2	true: итерация	true: уменьшение gcd
После четвертого прохода			2	2	false: завершение итерации	
После цикла			2		<-(итоговое значение GCD)	
Перед завершением программы	7	4			<-(итоговые значения полей)	

Рис. 10.4. *Точная трассировка выполнения функции-члена* `normalize()`

Здесь можно видеть два вызова функции `normalize()`. Один из них в функции `operator+()` применяет эту операцию к локальной переменной `temp`. В примере сложения 1/4 и 3/2 результатом будет `temp.nmr = 14`, `temp.dnm = 8`. Перед первым проходом цикла `while gcd = 14`, `value = 8`. На первом проходе (`14 > 8`)

gcd = 14 - 8 = 6, value = 8. На втором проходе (6 <= 8) gcd = 6, value = 8 - 6 = 2. После третьего прохода gcd = 4, value = 2. После четвертого прохода gcd = 2, value = 2, и цикл завершает работу. Итак, мы проследили работу алгоритма (см. рис. 10.4).

Второй вызов функции-члена normalize() — это вызов в общем конструкторе. Он необходим в том случае, если клиент создает объект вида:

```
Rational x(14,8);                    // допустимо, но некрасиво
```

Какова цель такого вызова функции-члена normalize()? В главе 9 достаточно много рассказывалось о том, что вид функций-членов значительно отличается от обычных глобальных функций, и для их вызова применяется разный синтаксис. (То, что является параметром в глобальной функции, становится получателем сообщения.) Как видно из примера, здесь нет целевого объекта (получателя сообщения), и функция вызывается подобно любой глобальной функции.

Если объект, который следует использовать в качестве адресата сообщения, явно не задан, то предполагается, что это — вызывающий функцию объект (если она не является глобальной). В данном примере адресатом сообщения будет объект x типа Rational, и функция normalize() будет использовать в своих алгоритмах поля nmr и dnm этого объекта.

Некоторым программистам не очень нравится, что вызовы глобальных функций и функций-членов синтаксически могут выглядеть похоже (без адресата сообщения). Вызывая глобальные функции, они применяют операцию глобальной области действия ::, а для вызова функций-членов того же класса используют указатель объекта this.

Почему они невзлюбили одно и то же обозначение функций-членов и глобальных функций? Ведь синтаксически это вполне корректно. Компилятор просматривает список функций-членов, определенных в классе. Если он находит соответствие, то проверяет интерфейс и генерирует вызов функции. Если соответствие не найдено, то компилятор повторяет поиск глобальных функций, доступных в данной области действия.

Все дело в удобстве этой записи для сопровождающего приложение программиста. Чтобы программа была понятнее (уменьшение сложности исходного кода означает повышение качества ПО), разработчику класса лучше указать, какие именно функции используются в классе — функции-члены или глобальные.



Внимание Нужно всегда стремиться найти способы сообщить максимум информации о разрабатываемом классе. Когда в вызове функции отсутствует адресат сообщения, укажите, что это вызов функции-члена класса (с помощью указателя this) или глобальной функции (через операцию ::).

В следующей версии класса Rational эти методы применяются для вызова функции normalize() в общем конструкторе и вызова глобальной функции labs, возвращающей в функции normalize() абсолютное значение типа long. Кроме того, функция normalize() перенесена из раздела public класса Rational в раздел private.

Если оставить данную функцию-член в разделе public, пришлось бы сообщить программисту, работающему над кодом клиента, что можно писать алгоритмы, приводящие к ненормализованным состояниям объектов Rational, а следовательно, нужно использовать данную функцию-член в коде клиента. Между тем эта функция добавлена в класс как раз с обратной целью — освободить клиент от обязанностей нормализации и перенести их на серверный класс. Если включить функцию в раздел public, это могло бы побудить программистов, создающих клиентскую часть, использовать ее и создать тем самым лишние зависимости. Такое решение столь же плохо, как включение в раздел public элементов данных класса.

Таким образом, перед создателем класса стоит важная задача изучения потребностей потенциальных клиентов и предоставления им максимально возможного сервиса, но не более того. Набор сервисов, предлагаемых классом, называется общедоступным интерфейсом класса. Данный интерфейс должен быть максимально узким, но при этом не лишать клиентский код сервисов, которые позволяют сделать программу понятной и независимой от внутренней архитектуры класса сервера.

```
class Rational {
  long nmr; dnm;                    // закрытые данные

void normalize()                    // закрытая функция-член
{ if (nmr == 0) { dnm = 1; return; }
  int sign = 1;
  if (nmr < 0) { sign = -1; nmr = ::labs(nmr); }          // для иллюстрации
  if (dnm < 0) { sign = -sign; dnm = ::labs(dnm); }
  long gcd = nmr, value = dnm;      // поиск наибольшего общего делителя (НОД)
  while (value != gcd) {            // остановить, когда найден НОД
    if (gcd > value)
       gcd = gcd - value;       // вычесть наименьшее значение из наибольшего
    else value = value - gcd; }
  nmr = sign * (nmr/gcd); dnm = dnm/gcd; }          // делитель положительный
public:
  Rational()                    // конструктор по умолчанию: нулевые значения
  { nmr = 0; dnm = 1; }
  Rational(long n, long d)      // общий конструктор: дробь как n/d
  { nmr = n; dnm = d;
    this->normalize(); }
Rational operator + (const Rational &x) const
{ return Rational(nmr*x.dnm + x.nmr*dnm, dnm*x.dnm); }
                              // ОСТАЛЬНАЯ ЧАСТЬ КЛАССА Rational
  } ;
```

Еще одно важное изменение, относящееся к классу Rational, касается функции operator+(). В данной версии исключен вызов operator() для передачи результатов вычислений конструктору Rational как аргументов. Применяемая в предыдущей версии класса Rational операторная функция обходится достаточно "дорого". Воспроизведем ее исходный код:

```
Rational Rational::operator + (const Rational &x) const
{ Rational temp;                    // n1/d1+n2/d2 = ((n1*d2)+(n2*d1))/(d1*d2)
  temp.nmr = (nmr * x.dnm) + (x.nmr * dnm);
  temp.dnm = dnm * x.dnm;           // например, 1/4 + 3/2 = 14/8
  temp.normalize();
  return temp; }
```

Сколько вызовов функций содержится во второй строке приведенного ниже примера для данной версии перегруженной операторной функции?

```
Rational a(1,4), b(3,2), c, d;
c = a + b;                          // c.nmr равно 7, c.dnm равно 4
```

Самый простой ответ: "Ни одного. Здесь только сложение двух дробей". Но это не так. Здесь нет операции сложения — используется перегруженная операторная функция (именно функция). Клиент можно переписать так, чтобы данная функция вызывалась явно:

```
Rational a(1,4), b(3,2), c, d; c = a.operator + (b);
                                    // c.nmr равно 7, c.dnm равно 4
```

Теперь каждому ясно, что здесь есть, как минимум, один вызов функции. Однако если посмотреть на ее тело, то можно увидеть вызов конструктора Rational при создании экземпляра объекта temp. А вызов функции normalize()? Так что на самом деле здесь три вызова функции.

Когда функция возвращает значение, тип которого соответствует определяемому программистом классу, создается новый неименованный объект данного класса, вызывается конструктор копирования, инициализирующий поля данного нового объекта значениями полей существующего объекта (в данном случае temp). Наконец, выполняется операция присваивания в выражении c = a.operator+(b). Это также эквивалентно вызову функции. Таким образом, можно насчитать уже пять вызовов.

Но и это еще не все. При достижении закрывающей фигурной скобки и завершении функции все локальные переменные (здесь — переменная temp) должны быть уничтожены. Что происходит при уничтожении экземпляра объекта? Правильно, вызывается деструктор. Кстати, после присваивания в клиенте неименованный объект, использовавшийся для возврата значения из функции, тоже должен быть уничтожен. При этом также вызывается деструктор. Итого — семь вызовов.

Хотелось бы сказать, что в новой версии операторной функции число вызовов сократится до двух, но увы... Можно устранить вызовы конструктора и деструктора для normalize() и конструктора копирования для возвращаемого значения. Вместо этого вводится вызов обобщенного конструктора и (внутри конструктора) функции normalize(). Таким образом, общее число вызовов будет равно пяти. Не очень большое достижение, но такие вещи обычно накапливаются.

При написании программы на C++ нужно научиться видеть скрытые вызовы функций. Здесь вызывается две функции, там — три, и программа будет выполнять достаточно много операций, реализуя при этом совсем немного полезных действий. Недаром программисты не любят возвращать из функций объекты, хотя в C++ это допустимо.



Осторожно! Научитесь распознавать в программе C++ вызовы конструкторов и деструкторов. Следует избегать лишних вызовов функций. Нужно делать это только для поддержки необходимого синтаксиса в клиенте.

Какие еще сервисы должен предоставлять класс Rational своим клиентам? Кроме operator+() нужно реализовать перегруженные операторные функции для трех других арифметических операций: operator-(), operator*(), operator/(). Подобно обсуждавшемуся в этой главе классу Complex, каждая арифметическая операторная функция должна возвращать значение типа класса. Результат можно присваивать другой переменной данного типа или использовать в качестве адресата сообщения в цепочке вызовов.

Численные алгоритмы часто требуют сравнения значений, и класс Rational — не исключение. Перегруженные операции сравнения должны возвращать true при совпадении значений и false (или 0) в противном случае.

```
bool Rational::operator == (const Rational &other) const
{ return (nmr * other.dnm == dnm * other.nmr); }

bool Rational::operator < (const Rational &other) const
{ return (nmr * other.dnm < dnm * other.nmr); }

bool Rational::operator > (const Rational &other) const
{ return (nmr * other.dnm > dnm * other.nmr); }
```

Аналогично перегружаются другие операции сравнения. Обратите внимание, что функции не изменяют своих параметров и целевых объектов, что и указывается с помощью const. Остается надеяться, что читатели это видят.

В листинге 10.5 показана реализация класса Rational и теста, демонстрирующего некоторые поддерживаемые им операции. Это хороший пример программы на C++ и использования перегруженных операторных функций, отражающий способы применения числовых типов данных и операций.

Листинг 10.5. Класс Rational и его тестирование

```
#include <iostream>
using namespace std;

class Rational {
  long nmr, dnm;                                    // закрытые данные
  void normalize();                                 // закрытая функция-член
public:
  Rational()                                  // конструктор по умолчанию: нулевые значения
  { nmr = 0; dnm = 1; }
  Rational(long n, long d)                          // общий конструктор: дробь как n/d
  { nmr = n; dnm = d;
    this->normalize(); }
Rational operator + (const Rational &x) const;      // цель - const
Rational operator - (const Rational &x) const;
Rational operator * (const Rational &x) const;
Rational operator / (const Rational &x) const;
  void operator += (const Rational &x);             // цель изменяется
  void operator -= (const Rational&);
  void operator *= (const Rational&);
  void operator /= (const Rational&);

bool operator == (const Rational &other) const;     // цель - const
bool operator < (const Rational &other) const;
bool operator > (const Rational &other) const;
void show() const;
  } ;                                               // конец спецификации класса

Rational Rational::operator + (const Rational &x) const
{ return Rational(nmr*x.dnm + x.nmr*dnm, dnm*x.dnm); }

Rational Rational::operator - (const Rational &x) const
{ return Rational(nmr*x.dnm - x.nmr*dnm, dnm*x.dnm); }

Rational Rational::operator * (const Rational &x) const
{ return Rational(nmr * x.nmr, dnm * x.dnm); }

Rational Rational::operator / (const Rational &x) const
{ return Rational(nmr * x.dnm, dnm * x.nmr); }

void Rational::operator += (const Rational &x)
{ nmr = nmr * x.dnm + x.nmr * dnm;                  // 3/8+3/2=(6+24)/16=15/8
  dnm = dnm * x.dnm;                                // n1/d1+n2/d2 = (n1*d2+n2*d1)/(d1*d2)
  this->normalize(); }

void Rational::operator -= (const Rational &x)
{ nmr = nmr * x.dnm - x.nmr * dnm;                  // 3/8+3/2=(6+24)/16=15/8
  dnm = dnm * x.dnm;                                // n1/d1+n2/d2 = (n1*d2-n2*d1)/(d1*d2)
  this->normalize(); }

void Rational::operator *= (const Rational &x)
{ nmr = nmr * x.nmr: dnm = dnm * x.dnm;
  this->normalize(); }

void Rational::operator /= (const Rational &x)
{ nmr = nmr * x.dnm; dnm = dnm * x.nmr;
  this->normalize(); }
```

```
bool Rational::operator == (const Rational &other) const
{ return (nmr * other.dnm == dnm * other.nmr); }

bool Rational::operator < (const Rational &other) const
{ return (nmr * other.dnm < dnm * other.nmr); }

bool Rational::operator > (const Rational &other) const
{ return (nmr * other.dnm > dnm * other.nmr); }

void Rational::normalize()                    // закрытая функция-член
{ if (nmr == 0) { dnm = 1; return; }
int sign = 1;
if (nmr < 0) { sign = -1; nmr = -nmr; }       // для иллюстрации
if (dnm < 0) { sign = -sign; dnm = -dnm; }
long gcd = nmr, value = dnm;                  // поиск наибольшего общего делителя
while (value != gcd) {                        // остановить, когда найден НОД
if (gcd > value)
gcd = gcd - value;                            // вычитание наименьшего числа из наибольшего
else value = value - gcd; }
nmr = sign * (nmr/gcd); dnm = dnm/gcd; }      // делитель положительный

void Rational::show() const
{ cout << " " << nmr << "/" << dnm; }

int main()
{ Rational a(1,4), b(3,2), c, d;
c = a + b;                                    // c.nmr равно 7, c.dnm равно 4
a.show(); cout << " +"; b.show(); cout << " =";
c.show(); cout << endl;
d = b - a;
b.show(); cout << " -"; a.show(); cout << " =";
d.show(); cout << endl;
c = a * b;                                    // c.nmr равно 3, c.dnm равно 8
a.show(); cout << " *"; b.show(); cout << " =";
c.show(); cout << endl;
d = b / a;
b.show(); cout << " /"; a.show(); cout << " =";
d.show(); cout << endl;
c.show();
c += b;
cout << " +="; b.show(); cout << " ="; c.show(); cout << endl;
d.show();
d *= b;
cout << " *="; b.show(); cout << " ="; d.show(); cout << endl;
if (b < c)
{ b.show(); cout << " <"; c.show(); cout << endl; }
return 0;
}
```

```
1/4 + 3/2 = 7/4
3/2 - 1/4 = 5/4
1/4 * 3/2 = 3/8
3/2 / 1/4 = 6/1
3/8 += 3/2 = 15/8
6/1 *= 3/2 = 9/1
3/2 < 15/8
```

Рис. 10.5. *Результат выполнения программы из листинга 10.5*

Многие разработчики чувствуют, что такой составной класс как Rational должен предоставлять клиентам строго определенный доступ к своим компонентам и предусматривать соответствующие функции set() и get().

```
long Rational::getNumer () const        // обратите внимание на const
{ return nmr; }

long Rational::getDenom () const
{ return dnm; }
```

```
void Rational::setNumer (long n)                    // не const
{ nmr = n; }

void Rational::setDenom (long d)
{ dnm = d; }
```

На самом деле, такое "усовершенствование" — лишь дополнительная работа для создателя класса и программиста клиентской части. В общем случае следует избегать предоставления клиенту доступа к деталям реализации (упорядоченного или нет). Если алгоритму клиента необходим такой доступ, можно попробовать поймать зубами пулю — сделать элементы данных общедоступными. Кроме того, структура рациональных чисел вряд ли будет меняться, и в классе всегда будет, как минимум, два поля. Изменение их имен также маловероятно, поскольку другие имена не дадут никаких дополнительных преимуществ. Если нужно ввести дополнительные поля, это не повлияет на существующий программный код, обращающийся непосредственно к числителю и знаменателю.



Внимание Убедитесь, что элементы данных определены как закрытые, а функции-члены — как общедоступные (public). Если локальные функции нужны только функциям-членам класса, то их нужно сделать закрытыми. Когда клиенту требуется доступ к элементам данных, а архитектура класса стабильная и вряд ли будет меняться, не следует создавать функций `get()` и `set()` — сделайте данные общедоступными.

Понятно, что сказанное идет вразрез с основными принципами инкапсуляции, сокрытия информации, переноса обязанностей на серверы и т. д. Возражать против закрытых данных на этом этапе — своего рода презрение к риску. Все равно, что смеяться над бытовыми проблемами.

Да, использовать в классе общедоступные данные — плохой тон. Это как захрапеть на деловой встрече. Но иногда такой вариант может сработать.



Внимание В классе с хорошо определенными и понятными элементами данных их можно сделать глобальными. В качестве примеров легко привести такие геометрические и алгебраические классы, как `Point`, `Rectangle`, `Line`, `Complex`, `Rational`.

Независимо от архитектуры этих геометрических и алгебраических классов их элементы данных вряд ли куда-нибудь исчезнут, а потому нет опасности изменения клиентской программы. Если нужно добавить дополнительные сервисы или функции-члены, сделать это нетрудно. Конечно, неразборчивое использование общедоступных элементов данных усложнит модификацию классов.

Параметры смешанных типов

Классы `Complex` и `Rational` — хорошие примеры определяемых программистом типов, которые эмулируют свойства встроенных числовых типов C++. С объектами этих типов можно работать при помощи того же набора операций, какой используется для обычных числовых переменных. При этом реализуется такая цель языка C++, как интерпретация определенных программистом типов в качестве встроенных типов C++.

И все же аналогия неполная. Можно применять к переменным числовых типов ряд операций, не применимых к объектам типа `Complex` или `Rational`. Примерами таких операций являются деление по модулю, поразрядные логические операции и сдвиги. Конечно, можно перегрузить эти операции подобно арифметическим

операциям и операциям сравнения, но смысл таких операций (каким бы он ни был) не будет интуитивно понятен программисту, создающему или сопровождающему клиентское приложение. Пример такого произвольного назначения смысла — `Complex::operator+()`, реализованный для вывода значений элементов данных объекта `Complex`. Вряд ли можно сразу сказать, что должно делать выражение `+x` с переменной `x` типа `Complex`.

Еще одна проблема с равнозначной интерпретацией объектов встроенных типов и типов, определяемых программистом, — это проблема неявного преобразования типов. C++ поддерживает безоговорочное преобразование типов. Например, для любых числовых типов следующие выражения будут синтаксически и семантически корректными:

```
c += b;          // OK для переменных b и c любых встроенных числовых типов
c += 4;          // OK для c любых встроенных числовых типов
```

Каким бы ни был числовой тип переменной `b`, он просто преобразуется к числовому типу переменной `c`. Независимо от типа переменной `c`, целое 4 преобразуется к этому типу. Если переменные `b` и `c` в приведенных примерах имеют тип `Rational`, то вторая строка дает ошибку. Чтобы она была синтаксически корректной, одна из данных функций должна быть доступна в области действия клиента.

```
void Rational::operator+=(int x);              // c+=4; есть c.operator+=(4);
void operator+=(Rational &r, int x);           // c+=4; есть operator+=(c,4);
```

Ни одна из этих функций в листинге 10.5 не реализована, что приводит к синтаксической ошибке в клиенте. Вот пример функции-члена, устраняющей такую ошибку:

```
void Rational::operator += (int x)             // целевой объект изменяется
{ nmr = nmr + x * dnm;                         // n1/d1 + n = (n1+n*d1)/d1
  this->normalize(); }
```

Обратите внимание, что если функции доступны (функция-член и глобальная функция с данными интерфейсами), то вторая строка в приведенном выше примере все равно будет ошибочной, на этот раз из-за неоднозначности вызова функции. Поскольку подходит любая из этих функций, компилятор не будет знать, какую из них вызывать.

Если переменные `b` и `c` в примере имеют тип `Complex`, то обе строки будут синтаксически корректны, так как в листинге 10.4 реализованы две версии функции `operator+=()`.

```
void Complex::operator += (const Complex &b);
void Complex::operator += (int b);
```

В коде клиента первая функция вызывается для первой строки примера, а вторая — для второй строки.

```
c += b;          // c.Complex::operator+=(b); аргумент Complex
c += 4;          // c.Complex::operator+=(4); аргумент integer
```

При этом разрешается проблема использования в выражениях операндов смешанного типа. Вторая перегруженная операторная функция работает не только для числовых аргументов, но и для символов, коротких и длинных целых, чисел с плавающей точкой и аргументов двойной точности с плавающей точкой. Согласно правилам преобразования аргументов, значение каждого из этих встроенных типов преобразуется в целое. Нет никакой необходимости перегружать для каждого из данных встроенных типов функцию `operator+=()`. Будет достаточно одной функции.

Но вздыхать с облегчением рано. А как насчет других операций: -=, *=, /=? Каждая из них требует другой перегруженной операторной функции с числовым параметром. И как насчет других операторных функций — operator+(), operator-(), operator*() и operator/()? Рассмотрим следующий пример для экземпляров объекта класса Rational:

```
c = a + b;                    // c = a.operator+(b);
c = a + 5;                    // ?? несовместимые типы ??
```

Вторая строка опять даст синтаксическую ошибку, поскольку перегруженная операция требует, чтобы в фактическом аргументе передавался объект типа Rational, а не значение встроенного числового типа. Однако все эти выражения — отнюдь не плод больного воображения. В алгоритмах числовые значения часто комбинируются с комплексными и рациональными числами. А как насчет сравнений? Хотелось бы иметь возможность сравнивать объекты типа Rational с целыми, а это ставит дополнительные вопросы.

Применяемое до сих пор решение было вполне законным, но сложным. Для каждой операторной функции с аргументом Rational (или объектом другого класса) приходится писать еще одну операторную функцию с аргументом long int. (На 16-разрядных машинах обычно достаточно int.)

Можно ли что-то с этим сделать? Да, C++ предлагает превосходный инструмент, позволяющий обойтись одним набором операторных функций (с параметром типа класса) и заставить принимать их фактические аргументы встроенных числовых типов.

Что же это за инструмент? Он дает возможность привести числовое значение к типу класса. Начнем с простого примера.

```
Rational c = 5;               // несовместимые типы?
```

Казалось бы, данная строка ошибочна. В главе 3 уже рассказывалось о принципах приведения типов — преобразовании значения одного из встроенных числовых типов в другой встроенный числовой тип. Конечно, такое преобразование осуществляется только для встроенных типов C++, а не для приведения значения встроенного типа к значению определенного программистом типа Rational. Но как бы выглядело такое приведение типов, если бы оно существовало? Можно было бы использовать тот же синтаксис, что и для числовых типов — имя типа указывать в круглых скобках, а в качестве его имени задавать тип, в который должно преобразовываться значение.

```
Rational c = (Rational)5;     // вот так может выглядеть приведение типа
```

В главе 3 вы уже видели две синтаксические формы приведения типа, одна из которых заимствована из языка C (эта форма и использована выше), а другая, в стиле C++, имеет вид, напоминающий функцию.

```
Rational c = Rational(5);     // вот так должно выглядеть приведение типа
```

Не напоминает ли это вызов конструктора? Как назвать функцию, порождающую значение типа класса? Разве не конструктором? Итак, эта функция выглядит как конструктор и ведет себя как конструктор. Стало быть, она и есть конструктор.

Но тогда возникает следующий вопрос — какой конструктор? Это просто. В главе 9 конструктор с одним параметром, тип которого отличается от типа класса, назывался конструктором преобразования. Теперь должно быть ясно, почему используется именно такое название. Данный конструктор преобразует значение (параметр) одного типа в значение типа класса. Чтобы приведенная выше строка была синтаксически корректной, нужно написать конструктор с одним параметром.

Что должен делать этот конструктор со своим параметром? Если, к примеру, значение параметра равно 5, то значение объекта Rational следует приравнять к 5, т. е. это будет 5/1. Если оно равно 7, то получается соответственно 7/1. Следовательно, значение параметра нужно использовать для инициализации числителя, а знаменателю, независимо от содержимого фактического аргумента, присвоить 1. В результате получается следующий конструктор:

```
Rational::Rational(long n)          // конструктор преобразования
{ nmr = n; dnm = 1; }               // инициализация целым числом
```

Данный конструктор вызывается каждый раз, когда при обращении к функции, ожидающей параметр Rational, задается числовой фактический аргумент. Класс Rational в этом случае должен выглядеть так:

```
class Rational {
  long nmr; dnm;              // закрытые данные
  void normalize();           // закрытая функция-член
public:
  Rational()                  // конструктор по умолчанию: нулевое значения 0/1
  { nmr = 0; dnm = 1; }
  Rational(long n)            // конструктор преобразования: дробь как n/1
  { nmr = 0; dnm = 1; }
  Rational(long n, long d)  // общий конструктор: дробь как n/d
  { nmr = n; dnm = d;
    this->normalize(); }
Rational operator + (const Rational &x) const
{ return Rational(nmr*x.dnm + x.nmr*dnm, dnm*x.dnm); }

     // ОСТАЛЬНАЯ ЧАСТЬ КЛАССА Rational
  } ;
```

Некоторые программисты предпочитают не писать несколько конструкторов, если всю необходимую работу может выполнить один конструктор с параметрами по умолчанию. Например, конструктор, который удобно использовать как общий конструктор, конструктор по умолчанию и конструктор преобразования, может иметь следующий вид:

```
Rational(long n=0, long d=1)          // конструктор: общий, преобразования,
                                      // по умолчанию
  { nmr = n; dnm = d;
    this->normalize(); }
```

Нужно понимать, что этот конструктор вызывается, когда клиент подставляет для инициализации объекта два аргумента, один или 0 аргументов. В таких случаях вместо отсутствующих аргументов при определении объектов Rational подставляются значения по умолчанию.

```
Rational a(1,4);              // Rational a = Rational(1,4); - два аргумента
Rational b(2);                // Rational b = Rational(2,1); - один аргумент
Rational c;                   // Rational c = Rational(0,1); - нет аргументов
```

Обратите внимание, что подставляемые в данном примере фактические аргументы имеют тип int, а конструктор ожидает тип long. Это не проблема: применяется неявное преобразование встроенных типов из int в long. Такое приведение типов доступно для встроенных числовых типов по умолчанию. В вызовах функций компилятор допускает не более одного преобразования встроенных типов и не более одного преобразования определенных программистом типов классов (вызовов конструктора преобразования).

При обработке выражений с операндами типа Rational компилятор сначала преобразует int в long, а затем в Rational. После такого преобразования компилятор генерирует вызов соответствующего оператора.

```
c = a.operator+(Rational((long)5));          // настоящий смысл c = a + 5;
```

Теперь приведенный выше программный код компилируется без оператора Rational::operator+(long). Создается временный объект Rational, вызывается конструктор преобразования, затем operator+(), а потом — деструктор Rational.

Итак, можно писать клиентскую часть с числовыми значениями во втором операнде, подставляя в первом операнде значение типа Rational:

```
int main()
{ Rational a(1,4), b(3,2), c, d;
  c = a + 5;                          // c = a.operator+(Rational((long)5));
  d = b - 1;                          // d = b.operator-(Rational((long)1));
  c = a * 7;                          // c = a.operator*(Rational((long)7));
  d = b / 2;                          // d = b.operator/(Rational((long)2));
  c += 3;                             // c.operator+=(Rational((long)3));
  d *= 2;                             // d.operator*=(Rational((long)2));
  if (b < 2);                         // if (b.operator<(Rational((long)2));
    cout << "Все работает\n";
  return 0; }
```

В листинге 10.6 показана новая версия класса Rational, поддерживающая смешанные типы в двоичных выражениях. Результат программы представлен на рис. 10.6.

```
1/4 + 5 = 21/4
3/2 - 1 = 1/2
1/4 * 7 = 7/4
3/2 / 2 = 3/4
7/4 += 3 = 19/4
3/4 *= 2 = 3/2
3/2 < 2
```

Рис. 10.6. *Результат программы из листинга 10.6*

Не забывайте, однако, что преобразование к типу Rational выполняется неявно, с помощью вызовов функций в конструкторе преобразования. Когда функция завершает работу, созданный для преобразования временный объект уничтожается с помощью вызова конструктора. (Для данного класса — это конструктор по умолчанию, поддерживаемый компилятором.) Помните пример с подсчетом количества вызовов функций в выражении? Две функции здесь, две функции там... (Можно было бы сказать: "Два доллара здесь, два доллара там".) Следовательно, эта версия класса Rational более медленная, чем версия, не использующая преобразования аргументов и предлагающая для аргументов каждого типа отдельную перегруженную операцию.

Листинг 10.6. Класс Rational поддерживает смешанные типы в выражениях

```
#include <iostream>
using namespace std;

class Rational {
  long nmr; dnm;                                  // закрытые данные
  void normalize()                                // закрытая функция-член
public:
  Rational(long n=0, long d=1)          // конструктор: общий, преобразования, по умолчанию
  { nmr = n; dnm = d;
    this->normalize(); }
Rational operator + (const Rational &x) const;          // цель - const
Rational operator - (const Rational &x) const;
Rational operator * (const Rational &x) const;
Rational operator / (const Rational &x) const;
void operator += (const Rational &x);                   // цель изменяется
void operator -= (const Rational &x);
void operator *= (const Rational &x);
void operator /= (const Rational &x);
```

```
bool operator == (const Rational &other) const;          // цель - const
bool operator < (const Rational &other) const;
bool operator > (const Rational &other) const;
void show() const;
  } ;                                                    // конец спецификации класса

Rational Rational::operator + (const Rational &x) const
{ return Rational(nmr*x.dnm + x.nmr*dnm, dnm*x.dnm); }

Rational Rational::operator - (const Rational &x) const
{ return Rational(nmr*x.dnm - x.nmr*dnm, dnm*x.dnm); }

Rational Rational::operator * (const Rational &x) const
{ return Rational(nmr * x.nmr, dnm * x.dnm); }

Rational Rational::operator / (const Rational &x) const
{ return Rational(nmr * x.dnm, dnm * x.nmr); }

void Rational::operator += (const Rational &x)
{  nmr = nmr * x.dnm + x.nmr * dnm;                      // 3/8+3/2=(6+24)/16=15/8
   dnm = dnm * x.dnm;                                    // n1/d1+n2/d2 = (n1*d2+n2*d1)/(d1*d2)
   this->normalize(); }

void Rational::operator -= (const Rational &x)
{  nmr = nmr * x.dnm - x.nmr * dnm;                      // 3/8+3/2=(6+24)/16=15/8
   dnm = dnm * x.dnm;                                    // n1/d1+n2/d2 = (n1*d2-n2*d1)/(d1*d2)
   this->normalize(); }

void Rational::operator *= (const Rational &x)
{  nmr = nmr * x.nmr; dnm = dnm * x.dnm;
   this->normalize(); }

void Rational::operator /= (const Rational &x)
{  nmr = nmr * x.dnm; dnm = dnm * x.dnm;
   this->normalize(); }

bool Rational::operator == (const Rational &other) const
{ return (nmr * other.dnm == dnm * other.nmr); }

bool Rational::operator < (const Rational &other) const
{ return (nmr * other.dnm < dnm * other.nmr); }

bool Rational::operator > (const Rational &other) const
{ return (nmr * other.dnm > dnm * other.nmr); }

void Rational::show() const
{ cout << " " << nmr << "/" << dnm; }

void Rational:: normalize() закрытая функция-член
{  if (nmr == 0) { dnm = 1; return; }
   int sign = 1;
   if (nmr < 0) { sign = -1; nmr = -nmr; }
   if (dnm < 0) { sign = -sign; dnm = -dnm; }
   long gcd = nmr, value = dnm;                          // поиск наибольшего общего делителя
   while (value != gcd) {                                // остановить, когда найден НОД
      if (gcd > value)
         gcd = gcd - value;                              // вычитание наименьшего члена из наибольшего
      else value = value - gcd; }
   nmr = sign * (nmr/gcd); dnm = dnm/gcd; }              // делитель положительный
```

```
int main()
{  cout << endl << endl;
   Rational a(1,4), b(3,2), c, d;
   c = a + 5;                                          // позднее обсудим c = 5 + a;
   a.show(); cout << " +"; << 5 << " ="; c.show(); cout << endl;
   d = b - 1;
   b.show(); cout << " -"; << 1 << " ="; d.show(); cout << endl;
   c = a * 7;
   a.show(); cout << " *"; << 7 << " ="; c.show(); cout << endl;
   d = b / 2;
   b.show(); cout << " /"; << 2 << " ="; d.show(); cout << endl;
   c.show();
   c +=3;
   cout << " *= " << 3 << " ="; c.show(); cout << endl;
   d.show();
   d *= 2;
   cout << " *= "; << 2 << " ="; d.show(); cout << endl;
   if (b < 2)
   { b.show(); cout << " < " << 2 << endl; }
   return 0;
   }
```

Такое неявное использование конструкторов преобразования поддерживается не только для перегруженных операций, но и для любой функции, включая функции-члены и глобальные функции с параметрами-объектами. Как уже упоминалось в главе 9, конструкторы преобразования наносят удар по системе строгого контроля типов в C++. Если числовое значение используется вместо объекта намеренно, то все в порядке. Если же это ошибка, то компилятор не сообщит о ней.

C++ предлагает замечательные средства предотвращения подобных ошибок, вынуждающие разработчика кода клиента указать сопровождающему приложение программисту, что он делает. Они предусматривают применение в конструкторе ключевого слова explicit:

```
explicit Rational(long n=0, long d=1)          // не может вызываться неявно
  { nmr = n; dnm = d;
    this->normalize(); }
```

Если конструктор объявляется как явный (explicit), то любой неявный его вызов даст синтаксическую ошибку.

```
Rational a(1,4), b(3,2), c, d;
c = a + 5;                          // синтаксическая ошибка: неявный вызов
c = a + Rational(5);                // OK: явный вызов
d = b - 1;                          // синтаксическая ошибка: неявный вызов
d = b - (Rational)1;                // OK: явный вызов
if (b < 1)                          // синтаксическая ошибка: неявный вызов
if (b < Rational(2))                // OK: явный вызов
  cout << "Все нормально\n";
```

Это очень хорошая идея: она дает разработчику класса большие возможности управления объектами классов, которые использует программист, отвечающий за клиентскую часть.

Классы, определяемые программистом (такие, как Complex и Rational), должны по возможности эмулировать поведение встроенных числовых типов. Применение числовых переменных в выражениях вместо объектов не является ошибкой — это вполне законные методы реализации вычислительных алгоритмов. В приведенном

выше примере одни строки помечены "OK", а другие — как синтаксическая ошибка. Необходимость каждый раз анализировать использование числовых операндов в клиенте приводит к получению эстетически менее привлекательной программы.

С этой точки зрения применение ключевого слова explicit для конструкторов таких классов, как Complex и Rational, возможно, избыточно.



Внимание Не используйте ключевое слово explicit в конструкторах числовых классов, реализующих перегруженные операторные функции. Применяйте их для классов, где использование вместо объектов класса аргументов встроенных типов в вызове функции даст ошибку и не является допустимым способом употребления класса.

Дружественные функции

Давайте снова вернемся к перегруженным операторным функциям и посмотрим, как можно использовать их для интерпретации определяемых программистом типов подобно встроенным числовым типам и что еще нужно сделать.

Начнем с оператора, в котором крайне желательно одинаково интерпретировать переменные встроенных и определяемых программистом типов. C++ поддерживает такой подход, но программист платит за это отказом от свободы выбора имен функции. Имя функции должно начинаться с ключевого слова operator с добавлением символа (или символов) встроенной операции C++, которую нужно применять к объектам данного класса.

Есть некоторые незначительные ограничения на то, что можно делать, а что нет. Например, допускается применять только существующие операции C++ (нельзя создавать собственные операции, не распознаваемые языком). Не предусматриваются изменения относительного приоритета операций, их ассоциативности, числа воспринимаемых операндов. Но это все мелочи. Если программист будет придерживаться своих обязанностей, то C++ выполнит свои. Он распознает построенные по описанным выше правилам выражения, где операции используются как вызовы функций.

C++ позволяет также вызывать перегруженные операторные функции подобно вызову других функций C++ — по имени функции (ключевое слово operator плюс символ операции), однако лишь немногие программисты прибегают к такому способу. Если уж нужно вызвать эту функцию как функцию, зачем вообще утруждать себя ключевым словом operator? Было бы удобнее воспользоваться свободой выбора имен и дать функции более содержательное имя, типа addComplex() или addToComplex(). В примерах данной главы в вызовах функций полные имена перегруженных операторных функций использовались с одной целью — показать внутренние механизмы программы C++. Каждое применение перегруженной операции в выражении представляет собой вызов функции (как минимум одной). Если используются локальные или возвращаемые объекты, то применение операций влечет за собой также вызов конструкторов и деструкторов этих объектов.

Как нередко бывает в реальной жизни, выигрыш здесь может быть большим, чем плата за него. Ничто не ограничивает программиста в действиях внутри функций, заголовок которых согласуется с правилами перегруженных операторных функций. Хорошим примером является перегруженная функция operator+() класса Complex из листинга 10.4. Что означает в клиенте следующее:

```
Complex x(20,40), y(30,50);        // определение, инициализация
+x; +y;                            // то же, что x.operator+(); y.operator+();
```

Если бы x и y были целыми, то смысл второй строки ясен: сохранение знака значения. Не очень интересная операция, но двух мнений тут быть не может. В случае объектов Complex это бывает все, что угодно. В нашем примере — вывод

содержимого элементов данных. Во многих классах операции, применимые к числам, к объектам применяться не могут. Интерпретация объектов как числовых значений позволяет создавать программы, которые трудно назвать интуитивно понятными (как использование плюса для вывода данных). Это серьезная опасность. (Ниже мы обсудим лучшие способы перегрузки операций для ввода и вывода объектов.)

Опять же, выигрыш может быть и меньше, чем плата за него. Перегруженные операторные функции просты, когда относятся к двум экземплярам объектов. Если же один операнд является экземпляром объекта (получатель сообщения), а другой — операндом числового типа, то возникает проблема: применение синтаксиса операции дает вызов перегруженной операторной функции с несовместимым типом аргумента.

В предыдущем разделе уже обсуждались два возможных решения этой проблемы. Один из них состоит в удвоении числа перегруженных операторных функций. Для каждой функции с параметром типа класса нужно писать перегруженную операторную функцию с тем же параметром числового типа. Это хорошее решение, но оно приводит к "раздуванию" класса и затрудняет его понимание.

Еще одно решение состоит в том, чтобы перегружать для каждой операции только одну функцию (с параметром типа класса) и создавать для данного класса конструктор преобразования, приводящий значение числового типа к значению типа класса. Когда операция применяется с двумя операндами типа класса, конструктор перед обращением к перегруженной операторной функции не вызывается. Если второй операнд (параметр функции) имеет числовой тип, то конструктор вызывается неявно (или явно при указании ключевого слова `explicit`) перед вызовом перегруженной операторной функции. При таком решении размер класса остается управляемым, но оно влечет за собой создание и уничтожение временных объектов класса при каждом использовании в фактических аргументах значений числовых типов. Это может повлиять на производительность программы. Например, первая строка следующего примера не вызывает никаких конструкторов преобразования, а вторая вызывает.

```
Rational a(1,4), b(3,2), c;
c = a + b;      // c = a.operator+(b); - совпадают, нет вызова конструктора
c = a + 5;      // c = a.operator+(5); - вызывается конструктор преобразования
```

Но и это еще не конец истории о смешанных типах в выражениях. А как насчет подобной последовательности операторов в клиенте? Сложение двух объектов `Rational` поддерживается непосредственно. Сложение объекта `Rational` и числа реализуется через дополнительный вызов конструктора преобразования, однако сложение числа и объекта `Rational` не поддерживается.

```
Rational a(1,4), b(3,2), c;
c = a + b;      // c = a.operator+(b); - совпадают, нет вызова конструктора
c = a + 5;      // c = a.operator+(5); - вызывается конструктор преобразования
c = 5 + a;      // синтаксическая ошибка: c = 5.operator+(a); невозможно
```

Выражение, использующее перегруженную операторную функцию-член, всегда означает передачу сообщения левому операнду. Следовательно, левый операнд должен быть экземпляром объекта, а не числом. Числу нельзя передать сообщение. Но с точки зрения равнозначной интерпретации объектов и чисел последняя строка в этом примере так же законна, как предыдущая. Таким образом, если следовать равноправной интерпретации встроенных типов и типов, определяемых программистом, такая операция должна поддерживаться.

Если нужно использовать функцию, интерфейс которой отличается от потребностей клиента, один из способов решения проблемы состоит в создании функции-оболочки. Это функция с тем же именем. Интерфейс ее отвечает требованиям клиента, а единственное назначение состоит в вызове функции, которую нужно

использовать в клиенте. В случае функции operator+() класса Rational функция-оболочка будет иметь то же имя и воспринимать в качестве первого параметра числовое значение.

```
Rational Rational::operator + (int i, const Rational &x) const
{ Rational temp1(i);                          // конструктор преобразования
  Rational temp2 = temp1.operator+(x);        // перегруженная операция
  return temp2; }
```

А еще лучше так:

```
Rational Rational::operator + (long i, const Rational &x) const
{ Rational temp(i);                           // конструктор преобразования
  return temp + x; }                          // вызов operator+(const Rational&);
```

Использовать данную функцию невозможно. Здесь участвуют три стороны: получатель сообщения, числовой параметр и параметр-объект. Как объединить все это в одном вызове функции?

```
Rational a(1,4), b(3,2), c;
c.operator+(5, b);                            // c + ?
```

Вызов данной перегруженной операторной функции означает передачу сообщения ее левому операнду. Значит, смысл последней строки кода таков: объект c, плюс что-то еще. Однако требуется сложить 5 и еще что-то, поместив результат в c. Следовательно, строка даст синтаксическую ошибку. Попробуем снова:

```
Rational a(1,4), b(3,2), c;
c = b.operator+(5, b);                        // c = b + ?
```

Если бы имя функции не содержало ключевого слова operator, то это могло бы сработать. Значение 5 преобразуется в Rational, складывается с объектом b, и результат копируется в объект c. Использование объекта b как получателя сообщения выглядит неуместным. Этот объект ничего не делает с операцией. Имя функции содержит ключевое слово operator, поэтому такой синтаксис также не подходит. Здесь должно участвовать две стороны, а не три.

На самом деле было бы хорошо избавиться от целевого объекта и вызывать функцию только с двумя параметрами.

```
Rational a(1,4), b(3,2), c;
  c = operator+(5, b);                        // c = 5 + b; ?
```

Помните о первых перегруженных операторных функциях, которые использовались в классе Complex в листинге 10.2? Они не были функциями-членами класса. Это глобальные функции. Чтобы приведенный пример заработал, нужно определить функцию-оболочку как глобальную функцию:

```
Rational operatop + (long i, const Rational &x)          // не член класса
{ Rational temp(i);          // вызов конструктора преобразования
  return temp + x; }         // вызов Rational::operator+(const Rational&);
```

Здесь не просто удалена операция области действия класса. Устранен также модификатор const, указывающий, что в теле функции не изменяются поля целевого объекта. Целевого объекта нет, а потому не надо показывать, что его поля не изменяются.

Это хорошее решение, но оно слишком ограниченное. Неплохо было бы задействовать данную функцию для других способов записи выражения, а не только для случая, когда первый операнд числовой. Способ обобщить функцию состоит в том, чтобы исключить объект Rational и использовать первый параметр вместо тела функции в конструкторе преобразования.

Когда для переопределения бинарной операции используются функции-члены, левый аргумент будет неявным в форме указателя.

```
Rational operatop + (const Rational &x, const Rational &y)
{ return x.operator+(y); }    // вызов Rational::operator+(const Rational&);
```

Это лишь один из примеров, где для явного вызова функции-члена класса необходимо использовать синтаксис вызова функции, а не синтаксис операции. Он позволяет глобальной функции operator+() с двумя параметрами вызывать функцию-член класса с одним параметром.

Отметим, что синтаксис выражения в этой функции неудачен. Он может быть интерпретирован как рекурсивный вызов глобальной функции operator+(), который определен здесь.

```
Rational operator + (const Rational &x, const Rational &y)
{ return x = y;}            //рекурсивный вызов operator+(): бесконечный цикл
```

Шаги разработки интерфейса для этой глобальной функции специально показаны здесь подробно. Многие программисты не привыкли писать на C++ один и тот же алгоритм как функцию-член и как глобальную функцию. Правила перехода сформулированы в главе 9: глобальная функция имеет один дополнительный параметр типа класса. Функция-член не содержит этого параметра, но использует объект аргумента как получателя сообщения. Данные различия нужно понимать.

Придется признать, что и эта конструкция имеет недостатки. Не хотелось обсуждать их одновременно с другими вопросами, а потому внимание намеренно на них не концентрировалось. Теперь пришло время заняться данной проблемой. Когда такой синтаксис операции используется в коде клиента, у компилятора есть две возможности интерпретировать выражение: вызвать либо функцию-член класса с одним параметром, либо глобальную функцию с двумя параметрами. Каждая функция обеспечивает законную интерпретацию выражения, будь то выражение с двумя экземплярами объектов или с одним экземпляром объекта и значением встроенного типа (при этом вызывается соответствующий конструктор преобразования). Конечно, если применяются два операнда встроенных типов, все однозначно: компилятор интерпретирует выражение как встроенную операцию, а не вызов перегруженной операторной функции.

```
Rational a(1,4), b(3,2), c;
c = a + b;          // неоднозначность: c = a.operator+(b);
                    // или c = operator+(a,b);?
c = a + 5;          // c = a.operator+(Rational(5));
                    // или c = operator+(a,Rational(5));
c = 5 + a;          // нет неоднозначности: c = operator+(Rational(5),a);
                    // а не 5.operator+(a);
c = 5 + 5;          // нет неоднозначности: встроенная операция сложения
```

Очень жаль, поскольку это противоречит общему алгоритму анализа компилятором смысла имени, описанного в главе 9. Для неоператорных функций компилятор сначала ищет функцию-член класса, а если не находит соответствия в области действия класса, ищет имя среди глобальных функций, известных в данном файле. Для операторных функций на такую удачу рассчитывать не приходится.

Чтобы исключить неоднозначность выражения, где оба операнда представляют объекты, можно устранить операторную функцию-член и реализовать алгоритм операции непосредственно в глобальной функции. Тогда у компилятора будет только один способ интерпретации выражения.

```
Rational operator + (const Rational &x, const Rational &y)  // нет Rational::
{ return Rational(y.nmr*x.dnm+x.nmr*y.dnm,y.dnm*x.dnm); }
                                                    // закрытые данные?
```

Хорошее решение, но слишком прямое, "в лоб". Оно предполагает непосредственный доступ к полям своих параметров, но сама функция находится вне области действия класса Rational, и, следовательно, не имеет право это делать. В результате функция компилироваться не будет.

C++ предлагает интересный способ обойти данное ограничение: использование дружественных функций (friend functions). Дружественная функция — это функция, не являющаяся членом класса, но имеющая те же права доступа к компонентам класса, что и любая функция-член класса. Обратите внимание: не права доступа к данным класса, а права доступа к компонентам класса, т. е. дружественная функция может так же легко обращаться к закрытым (или защищенным) функциям-членам класса, как и к закрытым (или защищенным) элементам данных.

Дружественной функцией может быть глобальная функция или функция-член другого класса. Есть ситуации, когда желательно разрешить доступ к компонентам класса всем функциям-членам другого класса. В таком случае другой класс определяется как "друг" этого класса. (Подробнее об этом рассказывается в главе 12.) Между тем чаще всего дружественные функции представляют собой глобальные функции. Если нужно определить функцию одного класса как дружественную функцию другого класса, нужно дважды подумать: очень часто все можно сделать гораздо проще.

Чтобы определить функцию, как дружественную функцию класса, ее прототип включается в спецификацию класса (как если бы это был его компонент), а перед прототипом указывается ключевое слово friend. Вот и весь фокус: во всех отношениях эта функция подобна функции-члену класса.

```
class Rational {
  long nmr; dnm;                              // закрытые данные
  void normalize()                            // закрытая функция-член
public:
  Rational(long n=0, long d=1 )               // конструктор: общий,
                                              // преобразования, по умолчанию
  { nmr = n; dnm = d;
    this->normalize(); }
  friend Rational operator + (const Rational &x) const Rational &y);
                                             // ОСТАЛЬНАЯ ЧАСТЬ КЛАССА Rational
                                             // нет необходимости в функциях
operator+()
  };
```

Мои последние заявления зашли слишком далеко. Между дружественной функцией и функцией-членом большая разница. Для вызова дружественной функции не нужно указывать целевой объект, как это делается при вызове функции-члена, однако она может обращаться к компонентам класса Rational как его функция-член. Следовательно, теперь вполне допустима такая версия функции:

```
Rational operator + (const Rational &x, const Rational &y)          // нет
Rational:: { return Rational(y.nmr*x.dnm+x.nmr*y.dnm,y.dnm*x.dnm); }
                                                          // да, закрытые данные
```

Замена функции-члена класса дружественной функцией снимает неоднозначность в коде клиента:

```
Rational a(1,4), b(3,2), c;
c = a + b;          // нет неоднозначности: c = operator+(a,b);
c = a + 5;          // нет неоднозначности: c = operator+(a,Rational(5));
c = 5 + a;          // нет неоднозначности: c = operator+(Rational(5),a);
c = 5 + 5;          // нет неоднозначности: встроенная операция сложения
```

Поддерживаются все три формы выражения с объектами Rational. Если нежелательны вызовы конструктора преобразования Rational, можно избежать его, определив три перегруженные операторные функции и три дружественные функции класса Rational.

```
class Rational {
  long mnr; dnm;                          // закрытые данные
  void normalize()                        // закрытая функция-член
public:
  Rational(long n=0, long d=1)            // конструктор: общий, преобразования,
                                          // по умолчанию:
  { nmr = n; dnm = d;
    this->normalize(); }
  friend Rational operator + (const Rational &x, const Rational &y);
  friend Rational operator + (const Rational &x, long y);
  friend Rational operator + (long x, const Rational &y);
                                           //ОСТАЛЬНАЯ ЧАСТЬ КЛАССА Rational
  };
```

Как уже упоминалось выше, можно использовать аналогичные методы множественной перегрузки с функциями-членами. Дружественные функции имеют перед функциями-членами преимущество, так как функции-члены могут поддерживать только те формы, где левый операнд представлен объектом Rational, а не числовой переменной. Поскольку вызов перегруженной операторной функции интерпретируется как передача сообщения левому операнду, поддержка такой формы потребовала бы от компилятора понимания следующего выражения:

```
c = 5.operator+(a);              // целое значение не может отвечать
                                 // на сообщения Rational
```

Дружественные функции позволяют более гибко комбинировать числовые операнды и объекты, так как левый операнд не обязательно должен быть объектом.

Аналогичный подход можно применить к операциям отношения. Функция-член класса с параметром-объектом поддерживает только выражения с операндами, представляющими экземпляры объектов. Если нужно поддерживать выражения с числовым значением в правом операнде, следует добавить операцию преобразования или другую перегруженную операторную функцию с числовым параметром. Тем не менее при этом все равно не поддерживаются выражения, где левый операнд — числовое значение, а правый — объект.

```
Rational a(1,4), b(3,2);
if (a < b) cout << "a < b\n";              // a.operator<(b);
if (a < 5) cout << "a < 5\n";              // a.operator<(5);
if (1 < b) cout << "1 < b\n";              // 1.operator<(b); нонсенс
if (1 < 5) cout << "1 < 5\n";              // встроенная операция неравенства
```

Для поддержки четвертой строки в данном фрагменте клиента можно добавить в программу перегруженную операторную функцию:

```
bool operator < (const Rational &x, const Rational &y)
{ return x.operator<(y); }
```

Аналогично арифметической операции применение глобальной функции и операторной функции-члена создает неоднозначность во второй и третьей строке примера.

```
Rational a(1,4), b(3,2);
if (a < b) cout << "a < b\n";          // a.operator<(b); или operator<(a,b);
```

```
if (a < 5) cout << "a < 5\n";          // a.operator<(5); или operator<(a,5);
if (1 < b) cout << "1 < b\n";          // нет неоднозначности: operator<(1,b);
if (1 < 5) cout << "1 < 5\n";          // встроенная операция неравенства
```

Для поддержки всех форм выражений отношения можно заменить каждую операторную функцию-член глобальной функцией, обращающейся непосредственно к элементам данных параметров. Чтобы подобный доступ был законным, следует определить такую глобальную операторную функцию как дружественную функцию класса:

```
class Rational {
  long nmr; dnm;                              // закрытые данные
  void normalize()                            // закрытая функция-член
public:
  Rational(long n=0, long d=1)                // конструктор: общий,
                                              // преобразования, по умолчанию:
  { nmr = n; dnm = d;
    this->normalize(); }
  friend Rational operator + (const Rational &x, const Rational &y);
  friend Rational operator - (const Rational &x, const Rational &y);
  friend Rational operator * (const Rational &x, const Rational &y);
  friend Rational operator / (const Rational &x, const Rational &y);
  friend bool operator < (const Rational &x, const Rational &y);
  friend bool operator > (const Rational &x, const Rational &y);
  friend bool operator == (const Rational &x, const Rational &y);
                                          // ОСТАЛЬНАЯ ЧАСТЬ КЛАССА Rational
  };
```

Такая конструкция устраняет неоднозначность и поддерживает все формы операций отношения с объектами в обоих операндах, только в правом операнде и только в левом операнде (самый трудный случай).

```
Rational a(1,4), b(3,2);
if (a < b) cout << "a < b\n";          // operator<(a,b);
if (a < 5) cout << "a < 5\n";          // operator<(a.Rational(5));
if (1 < b) cout << "1 < b\n";          // operator<(Rational(1),b);
if (1 < 5) cout << "1 < 5\n";          // встроенная операция неравенства
```

Как видно, операторная дружественная функция может делать ту же работу, что и операторные функции-члены, и даже более того. Единственные операции, которые не могут перегружаться как дружественные, это операции присваивания (`operator=()`), индекса (`operator[]()`), селектора (`operator->()`) и операция круглых скобок (`operator()()`). Такое ограничение необходимо, чтобы первый операнд можно было использовать как l-значение (адресат сообщения). Во всех ранее приведенных примерах данного раздела первый и второй операнды представляли r-значения.

Теперь рассмотрим арифметические операции присваивания. Ситуация здесь несколько другая, так как эти операции возвращают не значение, а `void`). Они модифицируют состояние целевого объекта. Поскольку они не возвращают нового значения класса `Rational`, конструктор, нормализующий состояние данного объекта, не вызывается. Следовательно, перед возвратом результата арифметические операции должны вызывать функцию `Rational::normalize()`:

```
void Rational::operator += (const Rational &x)                  // нет const
{ nmr = nmr * x.dnm + x.nmr * dnm; dnm = dnm * x.dnm;
  this->normalize(); }                              // нет вызова конструктора
```

Эта операция поддерживает выражения, где левый и правый операнды являются объектами (например, c+=b;). При наличии конструктора преобразования будут поддерживаться также выражения, где левый операнд — объект, а правый — одно из числовых значений (например, c+=5;).

```
Rational a(1,4), b(3,2), c;
c = a + b;          // c = operator+(a,b);
c += b;             // c.operator+=(b);
c += 5;             // c.operator+=(Rational(5));
5 += c;             // 5.operator+=(c); нонсенс, не так ли?
```

На первый взгляд, замена перегруженной операторной функции-члена глобальной перегруженной операторной функцией не даст особого выигрыша.

```
class Rational {
  long nmr, dnm;                              // закрытые данные
  void normalize();                           // закрытая функция-член
public:
  Rational(long n=0, long d=1)                // конструкторы: общий,
                                              // по умолчанию, преобразования
  { nmr = n; dnm = d;
    this->normalize(); }
  friend void operator += (Rational &x, const Rational &y);      // нет const
                                              // ОСТАЛЬНАЯ ЧАСТЬ КЛАССА Rational
} ;
```

Данная операция изменяет значение первого параметра. Вот почему этот параметр не имеет модификатора const.

```
void operator += (Rational &x, const Rational &y)                // нет const
{ x.nmr = x.nmr*y.dnm + y.nmr*x.dnm; x.dnm *= y.dnm;
  x.normalize(); }              // здесь у normalize() есть адресат сообщения
```

Помните о дружественной функции, обращающейся ко всем компонентам класса, а не только к элементам данных? Такая согласованная интерпретация компонентов класса здесь окупается: подобная операторная функция обращается к закрытым элементам данных своих параметров и закрытой функции-членуnormalize().

```
Rational a(1,4), b(3,2), c; long x = 5;
c = a + b;          // c = operator+(a,b);
c += b;             // operator+=(c,b);
c += 5;             // operator+=(c,Rational(5));
5 += c;             // константа не может использоваться как l-значение
x += c;             // operator+=(Rational(x),c); что это?
```

Сложение чего-либо с числовым литералом (значением-константой) даст синтаксическую ошибку. Для числовой переменной операция будет сложнее. Эта переменная может изменяться, особенно при передаче функции, у которой параметр-ссылка не имеет модификатора const.

Между тем данный аргумент не является объектом Rational, а потому требуется преобразование типа. Компилятор создает временный объект, вызывает для его инициализации конструктор преобразования Rational и передает значение x конструктору как аргумент. Что же теперь? Модифицировать этот временный объект внутри операторной функции (как параметр x) бесполезно, так как после завершения функции объект будет уничтожен и изменения не передадутся обратно в переменную x. Порядочный компилятор пометит это как синтаксическую ошибку.

```
5 + 1/4 = 21/4
1 - 3/2 = -1/2
7 * 1/4 = 7/4
2 / 3/2 = 4/3
7/4 += 3 = 19/4
4/3 *- 2 = 8/3
a < 5
1 < b
1 < 5
b*b-a == c-1 == 15/4
```

Рис. 10.7. *Результат программы из листинга 10.7*

Хотя в данном случае числовые типы и типы, определяемые программистом, нельзя интерпретировать одинаково, пример показывает, что пройден достаточно длинный путь к этой цели. Многие программисты предпочитают использовать глобальные операторные функции `friend`, а не функции-члены, так как их легче писать. Они симметрично интерпретируют свои операции.

В листинге 10.7 приведен пример реализации класса `Rational` с перегруженными операторными функциями, определенными как дружественные функции, а не как функции-члены. Результат программы представлен на рис. 10.7.

Многие программисты, вместо использования дружественных функций, бьются с реализацией операций как функций-членов. Основная причина — убеждение, что они нарушают инкапсуляцию, сокрытие информации и все другие хорошие вещи, которые обещает нам объектно-ориентированное программирование.

Листинг 10.7. Класс `Rational` с перегруженными операторными дружественными функциями, поддерживающими смешанные выражения

```
#include <iostream.h>

class Rational {
   long nmr; dnm;                          // закрытые данные
   void normalize()                        // закрытая функция-член
public:
Rational(long n=0, long d=1)               // конструктор: общий, преобразования, по умолчанию:
   { nmr = n; dnm = d;
     this->normalize(); }
friend Rational operator + (const Rational &x, const Rational &y);
friend Rational operator - (const Rational &x, const Rational &y);
friend Rational operator * (const Rational &x, const Rational &y);
friend Rational operator / (const Rational &x, const Rational &y);
friend void operator += (Rational &x, const Rational &y);
friend void operator -= (Rational &x, const Rational &y);
friend void operator *= (Rational &x, const Rational &y);
friend void operator /= (Rational &x, const Rational &y);
friend bool operator == (const Rational &x, const Rational &y);
friend bool operator < (const Rational &x, const Rational &y);
friend bool operator > (const Rational &x, const Rational &y);
void show() const;
   };                                      // конец спецификации класса

void Rational::show() const
{ cout << " " << nmr << "/" dnm; }

void Rational::normalize()                 // закрытая функция-член
{  if (nmr == 0) { dnm = 1; return; }
   int sing = 1;
   if (nmr < 0) { sign = -1; nmr = -nmr; }
   if (dnm < 0) { sign = -sign; dnm = -dnm; }
   long gcd = nmr, valut = dnm;            // поиск наибольшего общего делителя
   while (value != gcd) {                  // остановиться, когда НОД найден
      if (gcd > value)
         gcd = gcd - value;                // вычесть меньшее число из большего
      else value = value - gcd; }
   nmr = sign * (nmr/gcd); dnm = dnm/gcd; }  // знаменатель всегда положителен

Rational operator + (const Rational &x, const Rational &y)
{ return Rational(y.nmr*x.dnm + x.nmr*y.dnm, y.dnm*x.dnm); }
```

```
Rational operator - (const Rational &x, const Rational &y)
{ return Rational(x.nmr*y.dnm - y.nmr*x.dnm, x.dnm*y.dnm); }

Rational operator * (const Rational &x, const Rational &y)
{ return Rational(x.nmr * y.nmr, x.dnm * y.dnm,); }

Rational operator / (const Rational &x, const Rational &y)
{ return Rational(x.nmr * y.dnm, x.dnm * y.nmr); }

void operator += (Rational &x, const Rational &y)
{ x.nmr = x.nmr * y.dnm + y.nmr * x.dnm; x.dnm *= y.dnm;
  x.normalize(); }

void operator -= (Rational &x, const Rational &y)
{ x.nmr = x.nmr * y.dnm + y.nmr * x.dnm; x.dnm *= y.dnm;
  x.normalize(); }

void operator *= (Rational &x, const Rational &y)
{ x.nmr *= y.nmr; x.dnm *= y.dnm;
  x.normalize(); }

void operator /= (Rational &x, const Rational &y)
{ x.nmr = x. nmr * y.dnm; x.dnm = x.dnm *y.nmr;
  x.normalize(); }

bool operator == (const Rational &x, const Rational &y)
{ return (x.nmr * y.dnm == x.dnm * y.nmr); }

bool operator < (const Rational &x, const Rational &y)
{ return (x.nmr * y.dnm < x.dnm * y.nmr); }

bool operator > (const Rational &x, const Rational &y)
{ return (x.nmr * y.dnm > x.dnm * y.nmr); }

int main()
{  Rational a(1,4), b(3,2), c, d;
   c = 5 + a;
   cout << " " << 5 << " +"; a.show(); cout << " =";
   c.show(); cout << endl;
   d = 1 - b;                                          // operator-(Rational(1),b);
   cout << " 1 -"; b.show(); cout << " ="; d.show(); cout << endl;
   c = 7 * a;                                          // operator*(Rational(7),a);
   cout << " 7 *"; a.show(); cout << " ="; c.show(); cout << endl;
   d = 2 / b;                                          // operator/(Rational(2),b);
   cout << " 2 /"; b.show(); cout << " ="; d.show(); cout << endl;
   c.show();
   c += 3;                                             // operator+=(c,Rational(3));
   cout << " +=" << 3 << " ="; c.show(); cout << endl;
   d.show();
   d *= 2;                                             // operator*=(d,Rational(2));
   cout << " *=" << 2 << " ="; d.show(); cout << endl;
   if (a < 5) cout << " a < 5\n";                      // operator<(a,Rational(5));
   if (1 < b) cout << " 1 <b\n";                       // operator<(Rational(1),b);
   if (1 < 5) cout << " 1 <5\n";                       // встроенная операция неравенства
   if (d * b - a == c - 1) cout << " d*b-a == c-1 ==";
   (c - 1).show(); cout << endl;
   return 0;
   }
```

Действительно, частое использование дружественных функций делает программу запутанной и более трудной в сопровождении. Сомнений здесь быть не может. А как насчет разумного применения дружественных функций? Что можно считать разумным и что излишним в применение к ним?

Лучший способ ответить на этот вопрос — вспомнить основную цель использования классов в C++: классы нужны, поскольку при применении автономных глобальных функций, обращающихся к структурам данных, связь между функциями и данными существует только в представлении разработчика, его идеи не передаются сопровождающему приложение программисту. Кроме того, инкапсуляция в этом случае добровольна, и любая функция может обращаться к данным непосредственно, без функции доступа. Правильно? А еще мы хотели получить локальную область действия класса, чтобы имена функций и данных, используемых в одной части программы, не конфликтовали с именами в других ее частях. Помните этот список? Приходится повторять его достаточно часто, чтобы вы могли применять данные критерии для оценки качества ПО на C++.

Вооружившись этими критериями, посмотрим на конструкцию перегруженных операций, реализованных в виде функций в листинге 10.6.

```
class Rational {
  long nmr; dnm;                                    // закрытые данные
  void normalize()                                  // закрытая функция-член
public:
  Rational(long n=0, long d=1)                      // конструктор: общий,
                                                    // преобразования, по умолчанию
  { nmr = n; dnm = d;
    this->normalize(); }
Rational operator + (const Rational &x) const;      // целевой объект - const
Rational operator - (const Rational &x) const;
Rational operator * (const Rational &x) const;
Rational operator / (const Rational &x) const;
void operator += (const Rational &x);               // цель изменяется
void operator -= (const Rational &x);
void operator *= (const Rational &x);
void operator /= (const Rational &x);
bool operator == (const Rational &other) const;     // целевой объект - const
bool operator < (const Rational &other) const;
bool operator > (const Rational &other) const;
void show() const;
  };                                                // конец спецификации класса
```

Ясна ли здесь связь между функциями и данными? Да, эту связь обозначают открывающая и закрывающая фигурные скобки класса. Защищены ли данные от доступа из других функций (не членов)? Да, элементы данных определены как закрытые, к ним нельзя обращаться вне класса. Существует ли опасность конфликта имен между компонентами класса `Rational` и компонентами других классов? Нет, в других классах можно определять функции с именами вида `operator+()` и т. д.— конфликта не будет.

Похоже, неплохая конструкция. Теперь сравним ее с вариантом из листинга 10.7, где используются дружественные функции:

```
class Rational {
  long nmr; dnm;                                    // закрытые данные
  void normalize()                                  // закрытая функция-член
public:
Rational(long n=0, long d=1)                        // конструктор: общий,
                                                    // преобразования, по умолчанию:
  { nmr = n; dnm = d;
    this->normalize(); }
```

```
friend Rational operator + (const Rational &x, const Rational &y);
friend Rational operator - (const Rational &x, const Rational &y);
friend Rational operator * (const Rational &x, const Rational &y);
friend Rational operator / (const Rational &x, const Rational &y);
friend void operator += (Rational &x, const Rational &y);
friend void operator -= (Rational &x, const Rational &y);
friend void operator *= (Rational &x, const Rational &y);
friend void operator /= (Rational &x, const Rational &y);
friend bool operator == (const Rational &x, const Rational &y);
friend bool operator < (const Rational &x, const Rational &y);
friend bool operator > (const Rational &x, const Rational &y);
void show() const;
  };                                                    // конец спецификации класса
```

Видите, к чему я клоню? Список функций, связанных с данными, здесь присутствует и находится между открывающей и закрывающей фигурными скобками класса. Он виден не только разработчику класса, но и сопровождающему приложение программисту. Защищены ли данные от доступа из функций, находящихся вне фигурных скобок класса? Да, данные объявлены закрытыми, и любая функция, которой нужно к ним обращаться, должна быть объявлена в классе как функция-член или дружественная функция. А как насчет конфликта имен? Предположим, нужно реализовать перегруженную операторную функцию `operator+()` как функцию `friend` класса `Complex`. Будет ли это имя конфликтовать с именем функции `operator+()`, являющейся дружественной функцией класса `Rational`? Нет, у функции `operator+()`, относящейся к объектам класса `Complex`, другая сигнатура.

```
Complex operator + (const Complex &x, const Complex &y);
```

А как насчет дружественных функций, нарушающих инкапсуляцию, сокрытие информации и другие обещания объектно-ориентированного программирования? Данная конструкция с дружественными функциями во всех отношениях столь же хороша, что и конструкция с функциями-членами. В основном это дело вкуса. На мой взгляд, дружественные операции легче писать и проверять. Еще одно важное отличие в том, что глобальные операции поддерживают все виды выражений, а функции-члены — только те формы, где левый операнд является объектом, а не числовым значением.



Советуем Без колебаний используйте дружественные функции при реализации перегруженных операторных функций. Их легче проектировать, чем функции-члены, они поддерживают все три формы выражений в клиенте (когда оба операнда — объекты, левый — объект и только правый операнд — объект). Не применяйте дружественные функции, когда они усложняют программу.

Итоги

В данной главе мы рассмотрели такие тонкости C++, как перегруженные операторные функции. В отличие от средств C++, обсуждавшихся в предыдущих главах, перегруженные операторные функции не являются абсолютно необходимыми для написания качественного ПО.

Можно возразить, что за исключением таких классов, как `Rational`, `Complex` и им подобных, применение перегруженных операций скорее делает программу более запутанной, чем облегчает ее понимание. Причина в том, что большинство классов отличаются от числовых типов, и применить к ним числовые операции непросто.

Например, что может означать операторная функция `operator+()` и `operator<()` для класса `Employee` ("служащий")? Или для класса `Transaction` ("транзакция")? Конечно, можно придать этим операциям некоторый смысл, но он не будет интуитивно понятным и общепринятым. Возможно, программа будет выглядеть лучше, если назвать такие функции `giveRaise()` и `hasSeniority()` или дать им какие-то более подходящие для приложения имена.

Между тем перегруженные операции применяются не столь уж редко. Они особенно популярны в библиотеках языка C++, включая STL (Standard Template Library), поэтому нужно понимать, что они делают и как реализованы.

Очень важно приведенное здесь сравнение функций-членов и дружественных функций. Часто решения при разработке программы принимаются произвольно, без учета целей объектно-ориентированного программирования.

Не нужно рассматривать дружественные функции как нечто непостижимое. Используйте их, если они дают большую гибкость и делают программу понятнее. Но не переусердствуйте.

Глава 11

Конструкторы и деструкторы: потенциальные проблемы

Темы данной главы

- ✔ Передача объектов по значению
- ✔ Перегрузка операций для нечисловых классов
- ✔ Конструктор копирования
- ✔ Перегрузка операции присваивания
- ✔ Практические аспекты: способы реализации
- ✔ Итоги

Ранее рассматривались проблемы, связанные с одинаковой интерпретацией в программе C++ встроенных типов и типов, определяемых программистом.

В главе 10 обсуждались вопросы конструирования числовых классов, таких как `Complex` и `Rational`. Объекты данных классов — это полноценные экземпляры объектов. К ним применимо все, что возникает при работе с объектами. объявление класса, управление доступом к компонентам класса, проектирование функций-членов, определение объекта, его инициализация и передача объекту сообщений

Обсуждавшиеся ранее типы, определяемые программистом, были числовыми В отличие от целых чисел и чисел с плавающей точкой, они имеют более сложную внутреннюю структуру. Однако клиент обрабатывает такие объекты аналогично целым числам и числам с плавающей точкой. В клиентской программе их можно складывать, перемножать, сравнивать и т. д.

Обратите внимание на это утверждение. Под словами "такие объекты" понимаются объекты определяемых программистом типов. Что означает "целые числа и числа с плавающей точкой"? Здесь они сравниваются с типами, определяемыми программистом, а не с переменными целого типа или типа с плавающей точкой Лучше было бы сказать "встроенные" типы. Что же в таком случае означает "их" в последней фразе? Вероятно, то же, что и "такие объекты" выше. Но это не так, поскольку речь идет о работе с ними в коде клиента. Клиент не может работать с типами, определяемыми программистом: он манипулирует объектами подобного типа. Это экземпляры объектов или переменные. Они перемножаются, сравниваются и т. д.

Объекты определяемых программистом классов могут обрабатываться в коде клиента аналогично переменным встроенных типов. Таким образом, имеет смысл поддерживать для них перегруженные операции присваивания. Принцип C++, предусматривающий интерпретацию встроенных и определяемых программистом типов, работает и для этих классов. В данной главе рассматриваются перегруженные операции для классов, объекты которых можно складывать, перемножать, вычитать или делить. Например, для операций со строками текста в памяти может использоваться класс `String`. Из-за нечислового характера таких классов перегруженные операторные функции для них становятся искусственными. Например, с помощью перегруженной операции сложения можно либо реализовать конкатенацию строк, либо сравнить строки. Однако сложно найти способы для сложения или деления объектов `String`. Тем не менее, нужно знать, как работать с перегруженными операторными функциями для нечисловых классов.

Эти нечисловые классы имеют важное отличие: объекты одного класса могут использовать разные объемы данных. Объекты числовых классов всегда применяют одни и те же объемы памяти. Например, в классе `Rational` присутствуют два элемента данных одного размера — числитель и знаменатель.

В классе `String` объем текста, хранимый в одном объекте, отличается от объема текста в другом объекте. Если класс зарезервирует для каждого объекта один и тот же объем памяти, то вы столкнетесь с такими проблемами: непроизводительной тратой памяти (когда фактический, используемый объем памяти меньше зарезервированного места) или переполнения памяти (когда объект должен хранить слишком много текста). Эти две опасности подстерегают разработчиков классов, выделяющих для каждого объекта один и тот же объем памяти.

C++ разрешает эту проблему. В динамически распределяемой области или в стеке он выделяет фиксированный объем памяти для каждого объекта. Кроме того, при необходимости в динамически распределяемой области выделяется дополнительный объем памяти, который для каждого объекта может быть разным и даже может изменяться для одного и того же объекта за время его существования. Например, объект `String` может получить дополнительную память для сохранения текста, добавляемого к существующему объекту в результате конкатенации.

Для динамического управления памятью используются конструкторы и деструкторы. Неквалифицированное их применение может отрицательно повлиять на производительность программы. Кроме того, в этом случае может быть испорчено содержимое памяти и нарушена целостность программы, что несвойственно никакому другому языку, кроме C++. Каждый программист, работающий с C++, должен знать об этих опасностях.

Введем дополнительные концепции для класса `Rational` фиксированного размера (см. главу 10). В этой главе к классу `String` применяются концепции динамического управления памятью, что даст читателям возможность лучше понять связи между объектами и между переменными встроенных типов. Несмотря на все усилия интерпретировать их эквивалентно, они различаются.

Не пропустите материал данной главы. Опасность, связанная с конструкторами и деструкторами C++, вполне реальна, и нужно знать, как защищать себя, своего начальника и своих пользователей от собственных программ.

Передача объектов по значению

В главе 7 приводились аргументы против передачи объектов функциям как значений параметров или как параметров-указателей и рекомендовалась передача параметров по ссылке.

Говорилось о том, что передача по ссылке почти столь же проста, как передача по значению, но выполняется быстрее (для входных параметров, не модифицируемых функцией). Передача по ссылке происходит так же быстро, как передача по указателю, но синтаксис намного проще (для выходных параметров, не изменяемых при выполнении функции).

Отмечалось также, что при передаче по ссылке для входных и выходных параметров используется одинаковый синтаксис. Рекомендовалось для выходных параметров указывать ключевое слово const, показывая, что в результате выполнения функции параметр не изменяется. Если модификаторы не используются, то это должно показывать, что при выполнении функции параметр модифицируется.

В главе 7 можно было видеть, что не всегда следует возвращать значения-объекты, если это не вызвано необходимостью передать сообщения возвращаемому объекту (синтаксис цепочки выражений).

В соответствии с данным подходом, передача по значению должна быть ограничена передачей встроенных типов как входных параметров функций и возвратом из функций параметров встроенных типов. Почему это приемлемо для входных значений встроенных типов? Их передача по указателю усложняет программу и может ввести в заблуждение при ее чтении (можно подумать, что данный параметр изменяется в функции). Передача параметров по ссылке (с ключевым словом const) не очень трудна. Таким образом, для встроенных типов используйте передачу параметров по значению.

В последней главе много рассказывалось о методах программирования, с помощью которых были показаны не только достоинства или недостатки различных режимов передачи параметров, но и фактическая последовательность вызовов.

Кроме того, ранее говорилось об инициализации и присваивании. Хотя в том и в другом случае используется знак равенства, интерпретируются они по-разному. В этом разделе для демонстрации различий применяются отладочные операторы.

Из всех функций класса Rational оставим только эти три: normalize(), show() и operator+(). Обратите внимание, что перегруженная операция operator+() не является функцией-членом класса Rational. Это "дружественная" функция. Именно поэтому в начале абзаца говорится: "Из всех функций класса Rational", а не "из всех функций-членов класса Rational". Таким образом, подчеркивается, что функция friend во всех отношениях эквивалентна функции-члену класса. Она реализуется в том же файле, что и другие функции-члены, имеет те же права доступа к закрытым элементам данных и бесполезна для работы с любым другим классом, кроме Rational. От функций-членов ее отличает только синтаксис вызова, но для перегруженных операций у функций-членов и у функций friend синтаксис будет одинаков.

Листинг 11.1. Пример передачи параметров-объектов по значению

```
#include <iostream.h>

class Rational {
  long nmr; dnm;                                  // закрытые данные
  void normalize();                               // закрытая функция-член
public:
Rational(long n=0, long d=1)              // конструктор: общий, преобразования по умолчанию
  { nmr = n; dnm = d;
    this->normalize();
    cout << " создан: " << nmr << " " << dnm << endl; }
Rational(const Rational &r)
  { nmr = r.nmr; dnm = r.dnm;
    cout << " скопирован: "<< nmr << " " << dnm << endl; }
void operator = (const Rational &r)               // операция присваивания
  { nmr = r.nmr; dnm = r.dnm;
    cout << " присвоен: " << nmr << " " << dnm << endl; }
~Rational()                                       // деструктор
  { cout << " уничтожен: " << nmr << " " << dnm << endl; }
friend Rational operator + (const Rational x, const Rational y);
void show() const;
  } ;                                             // конец спецификации класса
```

```
void Rational::show() const
{ cout << " " << nmr << "/" << dnm; }

void Rational::normalize()                          // закрытая функция-член
{ if (nmr == 0) { dnm = 1; return; }
  int sign = 1;
  if (nmr < 0) { sign = -1; nmr = -nmr; }           // сделать оба положительными
  if (dnm < 0) { sign = -sign; snm = -dnm; }
  long gcd = nmr, value = dnm;                      // наибольший общий делитель (НОД)
  while (value != gcd) {                            // стоп, если найден НОД
    if (gcd > value
       gcd = gcd - value;                           // вычесть меньшее из большего
    else value = value - gcd; }
  nmr = sing * (nmr/gcd); dnm = dnm/gcd; }          // сделать dnm положительным

Rational operator + (const Rational x, const Rational y)
{ return Rational(y.nmr*x.dnm + x.nmr*y.dnm, y.dnm*x.dnm); }

int main()
{ Rational a(1,4), b(3,2), c;
  cout << endl;
  c = a + b;
  a.show(); cout << " +"; b.show(); cout << " ="; c.show();
  cout << endl << endl;
  return 0;
  }
```

В обобщенном конструкторе Rational добавлен отладочный оператор. Он будет выполняться при каждом создании и инициализации объекта Rational в начале функции main() и в функции operator+().

```
Rational::Rational(long n=0, long d=1)              // значения по умолчанию
  { nmr = n; dnm = d;                               // инициализация данных
    this->normalize();
    cout << "создан: " << nmr << " " << dnm << endl; }
```

Отладочный оператор вывода добавлен также в конструктор копирования. Этот оператор выполняется при инициализации объекта Rational с помощью элементов данных другого объекта Rational, например при передаче функции operator+() параметров по значению или когда эта функция возвращает объект Rational.

```
Rational::Rational(const Rational &r)          // конструктор копирования
  { nmr = r.nmr; dnm = r.dnm;                  // копирование элементов данных
    cout << " скопирован: " << nmr << " " << dnm << endl; }
```

Данный конструктор вызывается, когда аргументы типа Rational передаются по значению дружественной операторной функции operator+(). Между тем этот конструктор копирования не вызывается, когда operator+() возвращает значение объекта, так как перед возвратом из функции operator+() вызывается общий конструктор с двумя аргументами.

Конструктор не выполняет никаких осмысленных операций для класса Rational. Он добавлен только в целях отладки. Деструктор вызывается всякий раз при уничтожении объекта Rational.

Самая интересная функция здесь — это перегруженная операторная функция. Ее задача состоит в копировании элементов данных одного объекта Rational в компоненты данных другого объекта Rational. Чем она отличается от конструктора

копирования? На данном этапе, ничем. Отличаться будет возвращаемый тип. Конструктор копирования не должен его иметь. Здесь возвращается тип void.

```
void Rational::operator = (const Rational &r)          // присваивание
  { nmr = r.nmr; dnm = r.dnm;                          // копирование данных
  cout << " присвоен: " << nmr << " " << dnm << endl; }
```

Перегруженная операция присваивания — это операция с двумя операндами. Во-первых, она имеет один параметр типа класса, и это функция-член, а не функция friend. Она работает с двумя объектами: получателем сообщения и параметром. Во-вторых, такая двухместная операция всегда записывается между первым и вторым операндом. При сложении двух операндов записывается первый операнд, операция, затем второй операнд (например, a + b). При применении присваивания также записывается первый операнд, операция, затем второй операнд (например, a = b). В случае вызова функции объект a является получателем сообщения. В приведенной выше операции присваивания nmr и dnm принадлежат целевому объекту a. Объект b — это аргумент вызова функции. В данной операции присваивания r.nmr и r.dnm принадлежат фактическому аргументу b. Следовательно, синтаксисом вызова операторной функции будет a.operator = (b).

Данная операция возвращает void, поэтому она не может продолжить цепочку присваивания в клиенте, такую как a = b = c. Следовательно, возвращаемое присваиванием b = c (или b.operator = (c)) значение используется как параметр в присваивании a.operator = (b.operator(c)). Чтобы данное выражение было допустимым, операция присваивания должна возвращать значение типа класса (здесь — Rational). Поскольку операция присваивания спроектирована так, что возвращает тип void, цепочка операций будет помечена компилятором как синтаксическая ошибка. Для первого анализа операции присваивания это не важно. Цепочка присваиваний описывается ниже.

```
создан: 1 4
создан: 3 2
создан: 0 1

скопирован: 3 2
скопирован: 1 4
создан:     7 4
уничтожен: 1 4
уничтожен: 3 2
присвоен:  7 4
уничтожен: 7 4
1/4 + 3/2 = 7/4

уничтожен: 7 4
уничтожен: 3 2
уничтожен: 1 4
```

Рис. 11.1.
Результат выполнения программы из листинга 11.1

Результат программы из листинга 11.1 показан на рис. 11.1. Первые три сообщения "создан" — результат задания и инициализации трех объектов Rational в функции main(). Два сообщения "скопирован" выводятся в результате передачи потока данных перегруженной операторной функции operator+(). Следующее сообщение "создан" появляется при вызове конструктора Rational в теле функции operator+().

Все конструкторы вызываются в начале выполнения функции. Далее следует серия событий, когда выполнение достигает закрывающей фигурной скобки в теле функции, уничтожаются локальные и временные объекты. Первые два сообщения "уничтожен" генерируются при уничтожении локальных копий фактических аргументов (3/2 и 1/4) и вызове деструкторов для этих объектов. Объект, содержащий сумму параметров, не может уничтожаться перед использованием в операции присваивания. Следующее сообщение "присвоен" появляется в результате вызова перегруженной операции присваивания, а сообщение "уничтожен" — после вызова деструктора для созданного в теле функции operator+() объекта. Последние три сообщения "уничтожен" выводятся как следствие вызова деструкторов при достижении закрывающей фигурной скобки функции main() и уничтожении объектов a, b, c. Поскольку конструктор копирования не вызывается, сообщение "скопирован" не появляется.

Если добавить в интерфейс функции operator+() два амперсанда, последовательность событий будет другой.

```
Rational operator + (const Rational &x, const Rational &y)          // ссылка
{ return Rational(y.nmr*x.dnm + x.nmr*y.dnm, y.dnm*x.dnm); }
```

При программировании на C++ важно, чтобы разные части программы были согласованы. Здесь изменяется интерфейс прототипа функции и обновляется объявление функции в спецификации класса. (Не важно, какая это функция — функция-член или функция `friend`.) В данном случае несогласованность разных частей программы не смертельна — компилятор предупредит, что она содержит синтаксические ошибки.

Результаты выполнения программы из листинга 11.1 с функцией `operator+()` показаны на рис. 11.2. Видно, что отсутствуют четыре вызова функций: два параметра-объекта не создаются и два параметра-объекта не уничтожаются.



Советуем Избегайте передавать экземпляры объектов по значению. Это приводит к дополнительным вызовам функций. Передавайте параметры по ссылке и помечайте их в интерфейсе функции как объекты-константы, если применим модификатор `const`.

```
создан: 1 4
создан: 3 2
создан: 0 1

создан:     7 4
присвоен:   7 4
уничтожен: 7 4
1/4 + 3/2 = 7/4

уничтожен: 7 4
уничтожен: 3 2
уничтожен: 1 4
```

Рис. 11.2. *Результат выполнения программы из листинга 11.1 при передаче параметров по ссылке*

Далее покажем разницу между инициализацией и присваиванием в листинге 11.1. Здесь в выражении `c = a + b` осуществляется присваивание переменной `c`. Откуда известно, что это присваивание, а не инициализация? Потому что имя типа находится слева от `c`. Сам тип определяется ранее в начале функции `main()`. Данная версия `main()` создает и инициализирует объект с суммой `a` и `b`, а не задает и присваивает значение `c` в отдельных операторах.

```
создан: 1 4
создан: 3 2
создан: 7 4
1/4 + 3/2 = 7/4

уничтожен: 7 4
уничтожен: 3 2
уничтожен: 1 4
```

Рис. 11.3. *Результат выполнения программы из листинга 11.1 при передаче параметров по ссылке и использование инициализации объектов*

```
int main()
{ Rational a(1,4), b(3,2), c = a + b;
  a.show(); cout << " +"; b.show(); cout << " ="; c.show();
  cout << endl << endl;
  return 0; }
```

На рис. 11.3 показаны результаты выполнения программы из листинга 11.1 с данной версией функции `main()` и при передаче параметров по ссылке. Как видно, операция присваивания и конструктор копирования не используются. Это естественный результат перехода от передачи по значению к передаче по ссылке.

Позднее мы используем аналогичный метод, чтобы продемонстрировать разницу между инициализацией и присваиванием для класса `String`.



Советуем Различайте инициализацию объектов и присваивание им значений. При инициализации вызывается конструктор и не учитывается операция присваивания. При присваивании вызывается операция присваивания и обходится вызов конструктора.

Рекомендуем избегать передачи параметров-объектов по ссылке, различать инициализацию и присваивание. Нужно уметь прочитать программу и сказать: "Здесь вызывается конструктор, а используется присваивание". Тренируйте свою интуицию, чтобы легко выполнять подобный анализ.

Перегрузка операций для нечисловых классов

Расширение встроенных операций числовых классов — естественный процесс. Перегруженные операторные функции для этих классов аналогичны встроенным операциям. Неверная интерпретация их смысла программистами, занимающимися клиентом или сопровождающими программу, маловероятна. Похоже, что идея одинаковой интерпретации значений встроенных и определяемых программистом типов ведет к простой реализации.

Операции сложения, вычитания и другие могут применяться также к объектам нематематических классов, но их применение может показаться искусственными. Все это напоминает значки-пиктограммы для активизации команд в графическом пользовательском интерфейсе.

Сначала был интерфейс командной строки, и пользователям приходилось набирать длинные команды с параметрами, ключами, переключателями и т. д. Для упрощения ввода придумали также меню с текстовыми пунктами. Выбирая пункт меню, пользователь мог инициировать команду, не набирая ее. Другой вариант — оперативные клавиши. Если пользователь будет нажимать последовательность таких клавиш, то он активизирует команду непосредственно. Ему не надо отрываться от клавиатуры и проходить несколько меню с подменю. Наконец, появилась инструментальная панель с графическими командными кнопками. Нажимая на такую кнопку, пользователь активизирует команду. Ему не надо запоминать комбинацию клавиш. Значки на этих кнопках понятны: Open, Close, Cut, Print. При добавлении новых подобных значков они становятся менее понятными. Появляются кнопки New, Paste, Output, Execute и т. д.

Чтобы пользователь смог выучить значки, добавляются всплывающие подсказки. Пользовательский интерфейс становится более сложным, для приложений требуется больше места на диске и в оперативной памяти, для их создания необходимо активно работать по программированию. Перейдем к изучению операторных функций для нечисловых классов. Познакомимся с дополнительными правилами, написанием программного кода, текстом программы. Возможно, в коде клиента лучше применять вызовы обычных функций, чем "новомодных" перегруженных операций.

Класс String

Рассмотрим пример использования перегруженных операторных функций для нечисловых классов: операторную функцию для конкатенации текста.

Введем класс `String` с двумя элементами данных: указателем на динамически распределяемый массив символов и целым, задающим максимальное число допустимых символов, которые можно вставлять в массив в динамически распределяемой памяти. Библиотека C++ Standard Library содержит класс `String` (с первой буквой в нижнем регистре). Это более мощный класс, чем `String` в данных примерах, однако он сложнее.

Клиент может создавать объекты этого класса двумя способами: определяя максимальное число допустимых символов и задавая текстовое содержимое строки. Для спецификации числа символов требуется целочисленный параметр, для спецификации текста — текстовый массив. Типы данных параметров различаются, поэтому их нужно использовать в разных конструкторах. Каждый из этих конструкторов имеет один параметр, который отличается от типа класса, но преобразуется к значению типа класса, они называются конструкторами преобразования.

Первый конструктор преобразования с параметром, задающим длину строки, имеет по умолчанию нулевое значение аргумента. Если объект `String` создается с помощью этого значения по умолчанию (параметры не указываются), то длина текста для такого объекта будет равна нулю. В этом случае первый конструктор преобразования применяется как конструктор по умолчанию, например `String s;`.

Второй конструктор преобразования с задаваемым в параметре символьным массивом не имеет значения аргумента по умолчанию. Ввести его было бы несложно, например пустую строку, но тогда компилятору будет трудно интерпретировать вызов функции `String s;`. Какой конструктор со значением по умолчанию следует вызывать: первый (с нулевой длиной) или второй (с пустой строкой)?

Текущее содержимое строки может модифицироваться в клиенте с помощью вызова функции-члена `modify()`, задающей новое содержимое текста целевого объекта. Для доступа к содержимому объекта `String` используйте функцию-член `show()`. С ее помощью возвращается указатель на динамически распределяемую область памяти, которая выделяется для объекта. В клиенте его можно применять для вывода содержимого строки, ее сравнения с другим текстом и т. д. Листинг 11.2 показывает программу, реализующую класс `String`.

Листинг 11.2. Класс `String` с динамически распределяемой памятью

```
#include <iostream>
using namespace std;

class String {
  char *str;                          // динамически распределяемый символьный массив
  int len;
public:
  String (int length=0);              // конструктор преобразования/по умолчанию
  String (const char*);               // конструктор преобразования
  ~String ();                         // освобождение памяти
  void modify(const char*);           // изменение содержимого массива
  char* show() const;                 // возврат указателя.массива
} ;

String::String (const char* s)
{ len = strlen (s);
  str = new char [len+1];             // размер по умолчанию равен единице
  if (str==NULL) exit (1);            // проверка на успех
  str [0] = 0; }                      // пустая строка нулевой длины допустима

String::String(int length)
{ len = length;                       // определение длины входной строки
  str = new char[len+1];              // выделение памяти в динамической области
  if (str==NULL) exit(1);             // проверка на успех
  strcpy(str,s); }                    // копирование текста в динамически распределяемую память

String::~String()
{ delete str; }                       // возврат памяти в динамической области (не указателя)

void String::modify(const char a[])   // здесь не управления памятью
{ strncpy(str,a,len-1);               // защита от переполнения
  str[len-1] = 0; }                   // правильное завершение строки

char* String::show() const            // плохая практика, но допустимо
{ return str; }

int main()
{ String u("Проверка");
  String v("Ничего плохого не случится");
  cout << " u = " << u.show() << endl;          // результат OK
  cout << " v = " << v.show() << endl;          // результат OK
  v.modify("Давайте надеяться");                // ввод усекается
  cout << " v = " << v.show() << endl;
  strcpy(v.show(),"Привет");
  cout << " v = " << v.show() << endl;
  return 0;    }
```

Динамическое управление памятью

Первая строка первого конструктора преобразования задает значение элемента данных len, а вторая — значение элемента данных str, выделяя в динамически распределяемой области память соответствующего размера. Затем проверяется, успешно ли выделена память. В начало выделенной области памяти помещается нуль (символ '\0'). Для любой библиотечной функции C++ это текстовое содержимое выглядит как пустое, хотя там есть место для заданного клиентом числа символов.

Если клиент определяет объект String и не задает аргументов, то данный конструктор используется как конструктор по умолчанию, выделяющий в динамической области один символ при присваивающий пустой строке значение '\0'.

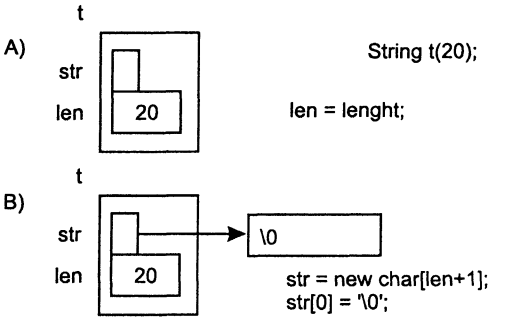


Рис. 11.4. *Диаграмма памяти для первого конструктора в листинге 11.2*

На рис. 11.4 показана схема памяти при выполнении каждого оператора конструктора для выражения:

```
String t(20);                // 21 символ в динамически распределяемой области
```

На рис. 11.4(A) представлена первая фаза выполнения конструктора, а на рис. 11.4(B) — вторая фаза. Прямоугольником обозначен объект t типа String с двумя элементами данных — указателем str и целым len. Эти элементы данных занимают одну область памяти, но указатель на меньший прямоугольник подчеркивает тот факт, что он не содержит данных для вычислений. Имя объекта t и имена элементов данных изображены вне прямоугольника объекта.

Часть A показывает, что после выполнения оператора len = length элемент данных len содержит значение 20 (содержит значение), а указатель str остается неинициализированным (ссылается куда угодно). Часть B демонстрирует, что после выполнения остальной части конструктора выделяется область в динамической памяти (21 символ), на нее ссылается указатель str, а первому символу присваивается значение 0. Читателям надо иметь диаграммы для всех фрагментов программ, где выполняются операции с памятью. Таким образом вы сможете лучше понять, как динамически управлять памятью.

Первая строка во втором конструкторе преобразования определяет длину заданной клиентом строки и устанавливает значение элемента данных len. Вторая строка присваивает значение элементу данных str и копирует внесенные клиентом символы в выделенную память. Библиотечная функция strcpy() копирует символы из массива-аргумента и добавляет завершающий нуль. На рис. 11.5 показаны этапы инициализации объекта для следующего оператора:

```
String u("Это тест");        // 15 символов, 16 символов в динамической области
```

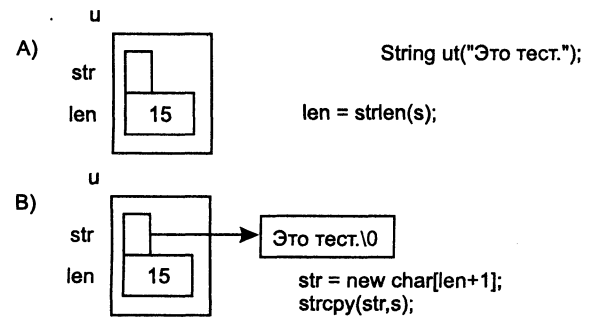


Рис. 11.5. *Диаграмма распределения памяти и второй конструктор преобразования из листинга 11.2*

Поддержание размера выделенной области для динамической памяти осуществляется тремя методами работы с элементами данных. Первый сохраняет общий объем динамически распределяемой памяти (число символов, плюс один). Второй отслеживает число полезных символов (как элемент данных) и при распределении памяти добавляет к этому значению единицу. Здесь используется второй подход, хотя трудно пояснить, почему он лучше первого.

Третий метод не поддерживает длину строки как элемент данных, а вычисляет ее оперативно с помощью вызова функции strlen(). Рекомендуем

его использовать, если длина требуется нечасто и не хочется добавлять для каждого строкового объекта дополнительный завершающий символ.

Поскольку динамическая память распределяется для каждого объекта String индивидуально, многие программисты чувствуют, что ее нужно рассматривать как часть объекта. При таком подходе объекты String представляются в клиенте как объекты переменной длины (в зависимости от размера выделенной динамической области памяти). Такая точка зрения имеет право на существование, но затрудняет объяснение работы конструкторов и деструкторов.

Используйте подход, проиллюстрированный диаграммами на рис. 11.4 и 11.5. Он отражает принцип C++, согласно которому класс — это схема, шаблон экземпляра объекта. Такая схема одинакова для всех объектов String. Каждый объект String имеет два элемента данных, а размер любого объекта String будет одним и тем же. В клиенте выполняется следующий оператор:

```
String t(20);      // для двух элементов данных распределяется память в стеке
```

Здесь для двух элементов данных объекта t выделяется память в стеке. Динамическая память распределяется функциями-членами String, выполняемыми для конкретного объекта. Для разных объектов String могут выделяться или освобождаться разные объемы динамической памяти, или они получают больше памяти без изменения своей "индивидуальности".

При таком подходе сами объекты String с памятью, выделяемой в динамической области, не изменяются.

```
String *p;                          // нет объекта String, создается
                                    // указатель в динамической области
p = new String ("Привет!");         // в динамической области два
                                    // элемента данных, плюс 4 символа
```

Здесь неименованный объект String, на который ссылается указатель p, получает числовой и символьный указатели в динамически распределяемой области. После создания объекта конструктор выделяет в динамической области место еще для 4 символов и устанавливает указатель str на эту память.

Данный подход удобен тем, что позволяет рассматривать все объекты как объекты одного класса и размера. При создании объекта реализуются два отдельных процесса: собственно создание объекта, причем всегда одного размера, и вызов конструктора. Он инициализирует элементы данных объекта, включая указатели, ссылающиеся на память в динамической области.

Деструктор освобождает память, выделяемую в динамической области. Он вызывается непосредственно перед уничтожением объекта. Когда объект уничтожается, освобождается для повторного использования память, выделенная для его элементов данных str и len. Если память для объекта выделялась в стеке, как для объектов u и v в функции main() из листинга 11.2, она возвращается в стек, если в динамически распределяемой области, как для неименованного объекта, на который ссылается указатель p, она снова возвращается в динамическую область. Во всех случаях память, освобождаемая деструкторами (на нее ссылается указатель str), возвращается в динамически распределяемую область до того, как исчезнут элементы данных len и str. В противном случае оператор delete str; был бы недопустимым.

Функция modify() изменяет содержимое динамически распределяемой области памяти. Чтобы содержимое памяти не оказалось запорченным, она использует библиотечную функцию strncpy(). В случае переполнения strncpy() не завершает строку нулевым символом. Это делается в конце функции. Если строка короче выделенной области памяти, такая операция кажется лишней. Нужно иметь в виду, что функция strncpy() в любом случае заполняет остаток строки нулями, и запись еще одного нуля не повлияет значительно на скорость выполнения программы.

С помощью функции modify() нельзя увеличить длину строки. Большинство конструкций класса String не позволяет программисту изменять содержимое объекта String. В этом случае для другого содержимого требуется создание и использование другого объекта. Для полномасштабных средств модификации потребовалось бы гораздо больше операторов. Для выполнения наших задач достаточно небольшой функции modify().

Функция show() возвращает указатель на динамически распределяемую память. Листинг 11.2 демонстрирует два варианта использования этой функции клиентом в main(). Первый вариант — вывод содержимого объекта String, получателя сообщения show(). Второй — это модификация содержимого объектов с помощью возвращаемого функцией show() значения. Оно применяется как выходной параметр в вызове функции strcpy() в клиенте. Первый вариант использования законен, а во втором случае возможности явно преувеличиваются. Сопровождающий приложение программист только запутается.

Один из первых языков программирования высокого уровня APL (A Programming Language) был очень сложным. Он использовался в основном для финансовых приложений. Набор символов в данном языке настолько велик, что кажется, будто для него нужна специальная клавиатура. Кроме того, он включает мощные операции с массивами и матричные операции. Занимающиеся APL программисты любят этот язык. Считается хорошим вкусом написать несколько строк на APL, показать другу и спросить: "Отгадай, что это значит?".

Однако таким специалистам не следует участвовать в коллективных проектах, где другие люди будут заниматься сопровождением их программ. Сегодня написание программы, которая требует дополнительных усилий для понимания, считается пустой тратой времени.

```
strcpy(v.show(),"Привет");          // плохая практика
```

Обратите внимание, что негодование автора направлено в основном на то, что сопровождающему приложение программисту понадобится проделать дополнительную работу для понимания программы. Действия этого оператора заключаются как будто не в оценке доступной в объекте размера динамически распределяемой памяти, а в том, чтобы досадить разбирающемуся в нем программисту. Это можно исправить с помощью разделения обязанностей между клиентом и сервером String:

```
int length = strlen(v.show());          // получить размер доступной памяти
strncpy(v.show(),"Привет!",length);     // перенос обязанностей "вверх"
```

Для объектов String, созданных вторым конструктором преобразования, значением length будет длина последней сохраненной строки. Она может быть меньше размера доступной области. Еще более важно то, что данный метод нарушает принцип переноса обязанностей с клиентов на серверы и сокрытия деталей операций с данными от клиента.

Здесь именно клиент выполняет операции с данными низкого уровня, хотя имена элементов данных String не используются. Если нужно защитить данные в динамически распределяемой области от порчи, то сервер должен предусматривать операции, определяющие размер доступной динамической памяти. Хорошим решением могло бы быть использование имени серверной функции, а не манипуляция с данными сервера и перенос обязанностей защиты памяти на сервер. Вы видели решение, которое помогает безупречно выполнить эту работу.

```
v.modifi("Hi there");       // тест доступного пространства
```

```
u = Это тест
v = Ничего плохого
v = Давайте надеяться
v = Привет!
```

Рис. 11.6. *Результат выполнения программы из листинга 11.2*

На рис. 11.6 показан результат программы из листинга 11.2. Он демонстрирует, что вызов функции modify() защищает динамическую память от переполнения, усекая данные клиента.

Использование указателя, возвращаемого функцией show(), не защищается. Пример порчи памяти, возможной при выполнении функции String::show().

```
char *ptr = v.show();          // необдуманный метод
ptr[200] = 'A';                // порча памяти
```

Если предпочтительнее цепочечная запись с объектами, используйте один оператор.

```
v.show()[200] = 'A';           // необдуманный метод, порча памяти
```

Это плохая практика программирования.

Защита данных объекта от клиента

C++ предусматривает способ защиты внутреннего содержимого объекта от клиента, который использует указатель, возвращаемый функцией-членом. Например, определим возвращаемое функцией show() значение как указатель на символ-константу (а не как указатель на неконстанту в листинге 11.2).

```
const char* String::show() const                    // хорошая практика:
                                                    // возвращается константа
{ return str; }
```

Теперь, если клиент попытается изменить содержимое динамически распределяемой памяти через возвращаемое функцией-членом show() значение, это будет помечено как синтаксическая ошибка.

```
strcpy(v.show(),"Привет!");    // ошибка, но не является неверной практикой
```

При такой конструкции серверного класса String для модификации состояния объекта клиент вынужден использовать функцию modify(). В результате клиент выражается в терминах вызовов серверных функций, обязанности защиты операций переносятся на серверный класс, и клиенту не нужно иметь дело с реализацией сервера (ограниченным объемом динамически распределяемой памяти).

Перегруженная операция конкатенации

Следующий шаг состоит в проектировании перегруженной операторной функции, соединяющей две строки — два объекта String. Содержимое второго объекта добавляется к содержимому первого объекта. Это означает, что клиент может использовать данную перегруженную операторную функцию следующим образом:

```
String u("Это тест. ");        // левый операнд
String v("Ничего плохого");    // правый операнд
u += v;                        // выражение: операнд, операция, операнд
```

После выполнения этого фрагмента программы содержимое объекта v должно остаться прежним, а содержимое объекта u будет заменено на "Это тест. Ничего плохого".

Если реализовать рассматриваемую операторную функцию как функцию-член, то объект u должен быть целевым объектом сообщения, а объект v — параметром в вызове функции. Реальный смысл такого фрагмента программы следующий:

```
u.operator+=(v);               // смысл u += v, -> u - это цель, а v - параметр
```

Следовательно, интерфейс данной функции должен включать для параметра модификатор const и не включать const для самой функции-члена. Возвращаемым типом может быть void. Это ограничивает использование операции в цепочках выражений, но для клиента такое ограничение нельзя считать серьезным.

```
void operator += (const String s);                  // конкатенация параметра
                                                    // с целевым объектом
```

Конечно, для передачи объектов по значению подобный вариант не приемлем, но здесь мы не будем акцентировать внимание на проблемах производительности. Кроме того, объект типа `String` имеет только два небольших элемента данных — символьный указатель и целое. Их копирование не займет много времени.

Алгоритм для конкатенации строки `String` включает следующие шаги:

1. Сложение длин обоих символьных массивов для определения общего числа символов.
2. Распределение динамической памяти для размещения символов и завершающего нуля.
3. Проверка распределения памяти. Отказ, если в системе не хватает памяти.
4. Копирование символов из целевого объекта в выделенную в динамической области память.
5. Конкатенация символов из объекта-параметра и размещение их в выделенной памяти.
6. Установка указателя `str` целевого объекта на выделенную область памяти.

Эти шаги (за исключением отказа от попытки выделить память в случае ее нехватки) вместе с реализующими их операторами C++ показаны на рис. 11.7. Чтобы было проще отслеживать события, здесь используются короткие строки.

В верхней части рисунка показаны два объекта типа `String` — объект `u` (с содержимым "Hi") и объект `v` (с содержимым "there!"). Часть A демонстрирует оба объекта после модификации поля `len` первого объекта, выделения памяти

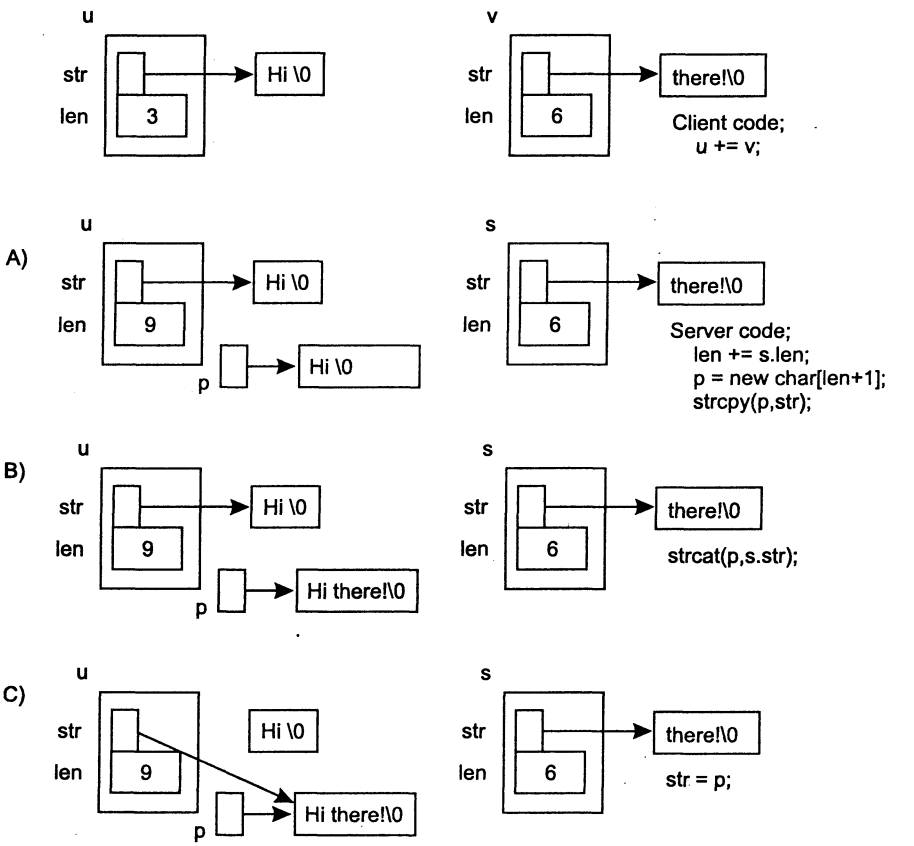


Рис. 11.7. *Диаграмма памяти для операторной функции конкатенации объектов `String`*

в динамически распределяемой области и копирования настоящего содержимого объекта u в динамическую память (шаги 1–4 алгоритма). Часть B показывает состояние динамически распределяемой памяти после шага 5. Часть C изображает состояние объектов после установки указателя str целевого объекта u на выделенную в динамической области память (шаг 6).

Если собрать все вместе, то получится:

```
void String::operator += (const String s)          // параметр-объект
{ char* p;                                  // локальный указатель
  len = strlen(str) + strlen(s.str);        // общая длина
  p = new char[len + 1];                    // распределение динамической памяти
  if (p==NULL) exit(1);                     // проверка на успех
  strcpy(p,str);                            // копирование первой части результата
  strcat (p,s.str);                         // конкатенация второй строки
  str = p; }                                // str указывает на новую память
```

Возможно, такой подробный разбор этапов алгоритма кажется лишним. Но вряд ли так думает большинство читателей. Многим операции с указателями кажутся сложными и непонятными.

Только опытные программисты могут заметить, что здесь не возвращается должным образом память, принадлежащая целевому объекту.

Изображение подобных схем — единственный способ для понимания механизмов распределения памяти и выявления ошибок. Лучше потратить несколько лишних минут на рисование и планирование, чем потерять часы на работу с отладчиком и другими сложными инструментами.

Конечно, такие схемы — всего лишь инструменты. Программисты должны использовать их, чтобы добиться понимания каждого оператора.

Предотвращение "утечек памяти"

Как уже упоминалось, на рис. 11.7 показано, что есть проблема с возвратом символьного массива в динамически распределяемой области памяти, на который ссылается целевой указатель str в начале вызова функции. Когда указатель str устанавливается на вновь выделенный сегмент памяти (куда ссылается локальный указатель p), массив становится недоступным. "Утечка памяти" — распространенная ошибка при управлении памятью. Чтобы предотвратить ее, нужно вернуть символьный массив до того, как указатель str будет установлен на новый выделенный массив.

```
void String::operator += (const String s)          // параметр-объект
{ char* p;                                         // локальный указатель
  len = strlen(str) + strlen(s.str);               // общая длина
  p = new char[len + 1];                    // распределение динамической памяти
  if (p==NULL) exit(1);                     // проверка на успех
  strspy(p,str);                            // копирование первой части результата
  strcpy(p,s.str);                          // конкатенация второй строки
  delete str;                               // возврат отведенной динамической памяти
  str = p; }                                // str указывает на новую память
```

Рис. 11.8 аналогичен рис. 11.7. Он показывает, что динамически распределяемый символьный массив, на который указывает целевой элемент данных str, исчезает в результате операции delete. Только после этого указатель str устанавливается на новый массив в динамически распределяемой области памяти.

Поговорим об "утечке памяти". Покажем, какие опасности связаны с использованием различных средств. При написании программы C++ никогда не нужно забывать о них. Как часто бывает, источником проблемы здесь является передача объектов в качестве значений параметров.

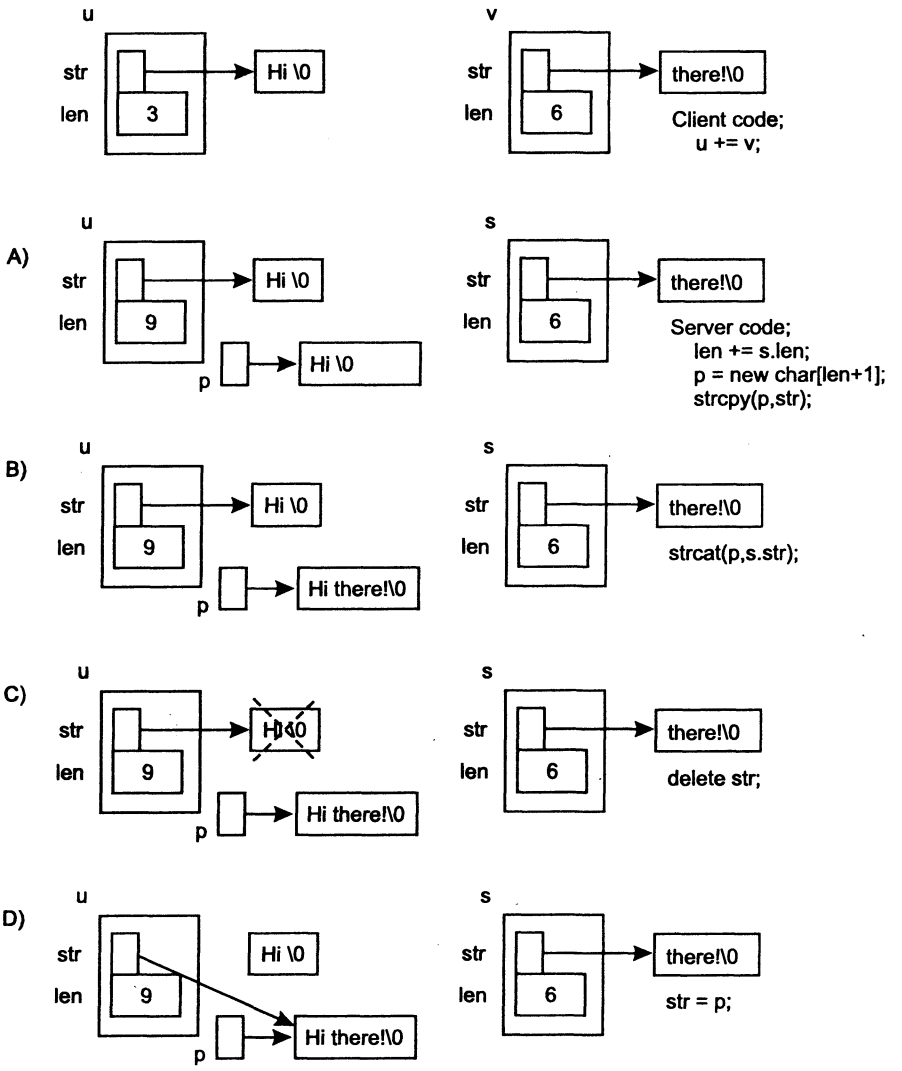


Рис. 11.8. *Диаграмма памяти для исправленной операторной функции конкатенации объектов* `String`

Защита целостности программы

Если фактический аргумент (объект или нет) передается по значению, то оно копируется в локальную автоматическую переменную в стеке. Копирование происходит поэлементно.

Для аргументов встроенных типов проблемы здесь не возникает, а для таких простых классов, как `Rational` или `Complex`, влияние на производительность программы будет незначительным. Трудности создаются для сложных классов, объекты которых требуют больших объемов памяти.

Если класс содержит элементы данных, представляющие собой указатели на динамически распределяемую область памяти, возникает угроза для целостности программы. Рассмотрим выполнение функции с параметром-значением в критические моменты этого процесса — в начале вызова функции и при ее завершении.

Когда при передаче по значению создается фактический объект-аргумент, вызывается подставляемый системой конструктор копирования. Он копирует элементы данных фактического аргумента в соответствующие данные своей локальной копии — формального параметра-объекта. При копировании элемента данных `str` указатель в формальном параметре-объекте получает значение, хранимое в указателе фактического объекта-аргумента, т. е. адрес динамической памяти, выделенной для фактического аргумента.

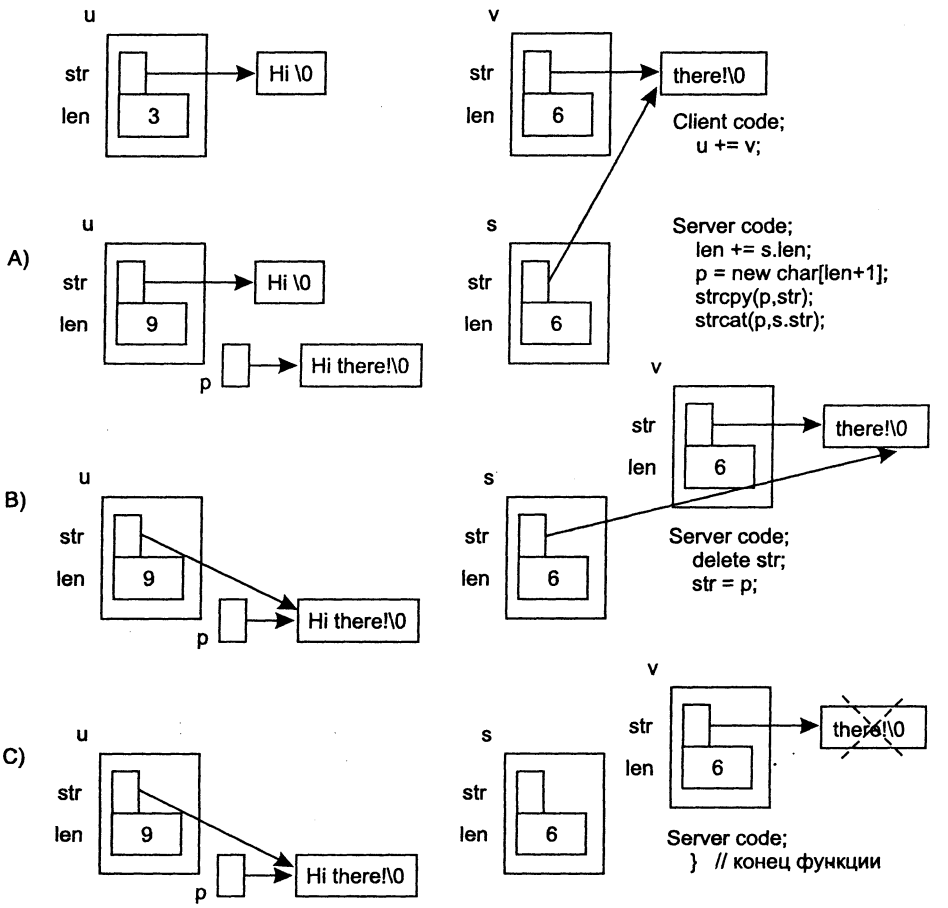


Рис. 11.9. *Диаграмма памяти для передачи объекта* `String` *по значению*

В результате указатели фактического аргумента и его локальной копии ссылаются на одну и ту же область в динамически распределяемой памяти. Каждый объект считает, что использует эту память эксклюзивно.

Данная ситуация изображена на рис. 11.9, где показан локальный объект с элементами данных, инициализированными значениями фактического аргумента v. Рис. 11.9(A) демонстрирует, что этот локальный объект v и фактический аргумент u совместно используют одну и ту же область динамически распределяемой памяти. На рис. 11.9(B) видно, что после распределения новой области динамической памяти, инициализации и замены существующей области динамической памяти в целевом объекте локальный объект s и аргумент u продолжают совместно использовать общую область динамически распределяемой памяти.

Теперь — о завершении функции. Когда функция достигает закрывающей фигурной скобки ее области действия и она завершает работу, локальный объект (`String s`) уничтожается. Обычно это означает, что исчезает (освобождается) память объекта (в данном случае указатель и целое значение). Между тем в C++ не уничтожается объект. Каждому уничтожению объекта предшествует вызов особой функции — деструктора.

При вызове деструктора происходит то, что написано в его коде: он возвращает сегмент памяти, на который ссылается указатель объекта.

```
String::~String()
{ delete [] str; }                    // возврат динамической памяти,
                                      // на которую ссылается указатель
```

Рис. 11.9(C) показывает состояние локального объекта s и фактического аргумента v после вызова деструктора и перед уничтожением локального объекта. Он демонстрирует, что локальный объект и фактический аргумент теряют свою динамически распределяемую память. (Освобождается память, на которую ссылается указатель str.) Конечно, это действие не влияет на состояние целевого объекта, так как он не уничтожается. При завершении работы перегруженной операторной функции целевой объект находится в том же состоянии, что и при предыдущем обсуждении (см. рис. 11.8). Клиент даст корректные результаты.

```
String u("Hi "); String v("there!");
u += v;
cout << " u = " << u.show() << endl;                    // выводит "Hi there!"
```

Между тем возвращаемая конструктором память при уничтожении формального параметра s уже не принадлежит данному объекту. Она относится к фактическому аргументу, т. е. к определенному в пространстве клиента объекту v. После вызова функции объект клиента, который используется как фактический аргумент для передачи по значению, теряет свою динамически распределенную память. Использовать ее после данного вызова в клиенте будет ошибкой.

```
String u ("Hi "); String v("there!");
cout << " u = " << u.show() << endl;                    // выводит "Hi "
cout << " v = " << v.show() << endl;                    // выводит "there!"
u += v;
cout << " u = " << u.show() << endl;                    // выводит "Hi there!"
cout << " v = " << v.show() << endl;                    // выводит все, что угодно
```

Не нужно проверять значение объекта v, который только что отображен на экране и использован как r-значение в вызове функции operator+=(). Это сделано здесь только потому, что проблема с данной реализацией известна заранее. Ясно, что у объекта должно быть то же значение, что и при использовании в качестве операнда в выражении u += v. В большинстве случаев (но не во всех) в C++ работает интуиция программиста. Поэтому нужно выработать альтернативную интуицию. Стоит еще раз повторить это. Даже в таком невинном на первый взгляд клиентском коде значением (текстом) в объекте v может быть все, что угодно, и любое использование данного объекта, предполагая, что он имеет то же состояние, что и прежде, будет безрассудным.

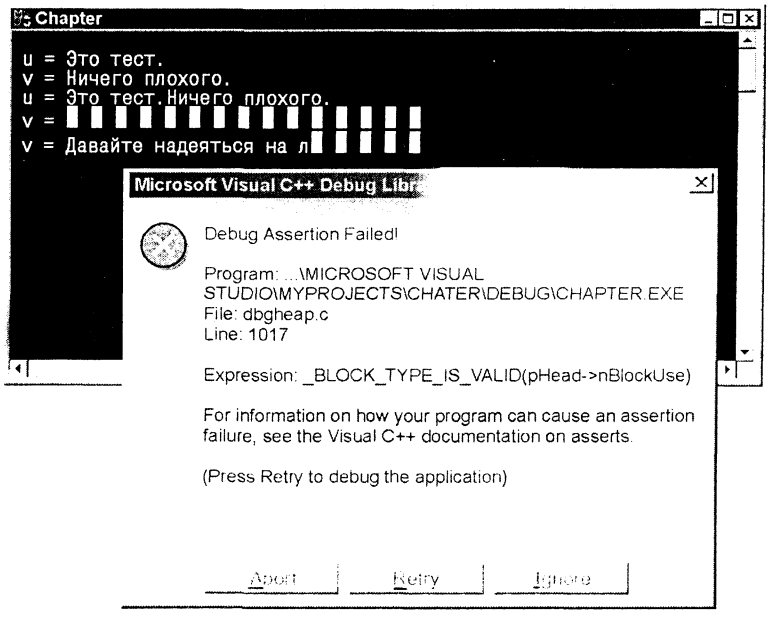


Рис. 11.10. *Результат выполнения программы из листинга 11.3*

Есть еще одна скобка, завершающая область действия. Обращайте внимание на все фигурные скобки, ограничивающие области действия. Они выполняют немало работы. Когда клиент достигает закрывающей фигурной скобки, и область действия завершается, для всех локальных объектов, включая объект v, который использовался как фактический аргумент при обращении к функции, вызываются деструкторы класса. Деструктор пытается освободить

память, на которую указывает элемент данных str. Между тем, эта память уже возвращена системе. При разработке языка можно было бы оформить такой вызов как операцию "no op". В C++ повторное использование операции delete с тем же указателем запрещено. Это ошибка.

К сожалению, "ошибка" не означает, что компилятор сообщит о синтаксической ошибке, которую можно исправить. Разработчик компилятора не следит за ходом выполнения программы и не выявляет ошибки программиста — анализируется лишь синтаксическая корректность кода. Это также не означает, что программа компилируется, выполняется и дает повторяющиеся некорректные результаты. Все зависит от платформы. На поведение приложения влияет операционная система. Система может аварийно завершить работу.

Листинг 11.3 демострируетет полную программу, реализующую такую плохую архитектуру. Вывод программы показан на рис. 11.10.

Листинг 11.3. Перегруженная функция конкатенации с параметром-значением

```
#include <iostream>
using namespace std;

class String {
   char *str;                          // динамически распределяемый символьный массив
   int len;
public:
   String (int length=0);              // конструктор преобразования по умолчанию
   String(const char*);                // конструктор преобразования
   ~String ();                         // освобождение памяти
   void operator += (const String);    // конкатенация с другим объектом
   void modify(const char*);           // изменение содержимого массива
   const char* show() const;           // возврат указателя массива
   } ;

String::String(int length)
{  len = length;
   str = new char[len+1];
   if (str==NULL) exit(1);             // проверка на успех
   str[0] = 0; }                       // пустая строка нулевой длины - OK

String::String(const char* s)
{  len = strlen(s);                    // определение длины входного текста
   str = new char[len+1];              // выделение достаточной памяти в динамической области
   if (str==NULL) exit(1);             // проверка на успех
   strcpy(str,s); }                    // копирование входной строки в динамически
                                       // распределяемую память

String::~String()
{ delete str; }                        // возврат памяти в динамической области (не указателя)

void String::operator += (const String s)        // передача по значению
{  len = strlen(str) + strlen(s.str);            // общая длина
   char *p = new char[len + 1];        // выделение достаточного объема памяти
   if (p==NULL) exit(1);               // проверка на успех
   strcpy(p,str);                      // копирование первой части результата
   strcat (p,s.str);                   // добавление второй части результата
   delete str;                         // важный шаг
   str = p; }                          // теперь p может исчезнуть

const char* String::show() const       // защита данных от изменений
{ return str; }
```

```
void String::modify(const char a[])              // здесь нет управления памятью
{ strncpy(str,a,len-1);                          // защита от переполнения
  str[len-1] = 0; }                              // правильное завершение строки

int main()
{ String u("Проверка");
  String v("Ничего плохого не случится");
  cout << " u = " << u.show() << endl;           // результат OK
  cout << " v = " << v.show() << endl;           // результат OK
  u += v;                                        // u.operator+=(v);
  cout << " u = " << u.show() << endl;           // результат OK
  cout << " v = " << v.show() << endl;           // результат не OK
  v.modify("Давайте надеяться на лучшее.");      // порча содержимого памяти
  cout << " v = " << v.show() << endl;           // ????
  return 0;
  }
```

Обратите внимание, что все неприятности происходят при завершении функции. Первая проблема возникает, когда завершается серверная перегруженная операторная функция `operator+=()` и вызывается деструктор для формального параметра — фактический аргумент `v` теряет свою динамически распределяемую память. Вторая неприятность случается, когда завершает работу клиент `main()` и объект `v` оказывается вне области действия. В этом случае память освобождается повторно.

В C++ повторное освобождение динамически распределяемой области памяти считается ошибкой. Удаление же указателя `NULL` не будет ошибкой. Это "пустая операция". Некоторые программисты пытаются решить проблему, присваивая в деструкторе указателю на динамическую память значение `NULL`.

```
String::~String()
{ delete str;              // возврат динамически распределяемой памяти
str = 0; }                 // установить в null, чтобы избежать
                           // двойное освобождение памяти
```

Устанавливаемый в нуль указатель принадлежит объекту, который через несколько микросекунд будет уничтожен. Вы могли бы установить в нуль второй указатель, ссылающийся на ту же память, но он недоступен в деструкторе, выполняемом в другом объекте. Если бы даже такой способ сработал, то можно было бы лишь предотвратить "ошибку" а не восстановить некорректно удаленную память.

Переход из пункта A в пункт B

Внимательно относитесь к управлению динамической памятью в программах. Даже если программы выполняются на машине правильно, вовсе не очевидно, что программа корректна.

В течение месяца или нескольких лет программа может работать корректно. Однако после установки каких-то других приложений или перехода на следующую версию Windows изменяется характер использования памяти, и программа завершается аварийно или дает некорректные результаты. Это может остаться незамеченным, т. к. ранее программа всегда работала корректно. Что же делать? Ругать Microsoft, поскольку вы только что модернизировали операционную систему? Но Microsoft тут ни при чем! Это ошибка программиста, забывшего включить один символ & в интерфейс перегруженной операторной функции, такой как `operator+=()`.

Вот как должна выглядеть эта функция. Она не передает свой параметр-объект по значению. Он передается по ссылке.

```
void String::operator += (const String &s)        // параметр-ссылка
{ len = strlen(str) + strlen(s.str);              // общая длина
  char *p = new char[len + 1];          // выделение достаточного объема памяти
  if (p==NULL) exit(1);                 // проверка на успех
  strcpy(p,str);                        // копирование первой части результата
  strcat(p,s.str);                      // добавить вторую часть результата
  delete str;                           // важный шаг
  str = p; }                            // str указывает на новую память
```

```
u = Проверка
v = Ничего плохого не случится
u = Проверка Ничего плохого не случится
v = Ничего плохого не случится
v = Давайте надеяться на л

Press any key to continue_
```

Рис. 11.11.
Результат программы из листинга 11.3 с операцией конкатенации, передающей параметр по ссылке

На рис. 11.11 представлен результат программы из листинга 11.3 с функцией конкатенации, передающей параметр по ссылке.

Попробуйте выполнить эту программу, поэкспериментируйте с ней. Не поддавайтесь желанию передавать объекты по значению, если в том нет абсолютной необходимости.

Конечно, разочаровывает то, что изменить программу можно с помощью добавления или удаления всего одного символа в исходном коде программы (амперсанда). Обратите внимание, что обе версии синтаксически корректны. Компилятор не показывает, что могут возникнуть проблемы.

Передавать параметры-объекты по значению — все равно, что ездить на танке. Вы попадете, куда хотите, но при этом натворите немало бед. Не спешите передавать объекты по значению, если это не является срочным решением.



Осторожно! Не передавайте функциям объекты по значению. Если объекты имеют внутренние указатели и могут работать с динамически распределяемой областью памяти, то не стоит даже и думать о передаче объектов по значению. Отправляйте их по ссылке. Используйте модификатор `const`, если функция не изменяет состояние объекта-параметра и целевого объекта.

Конструктор копирования

Поговорим о копировании объекта, элементы данных которого представляют собой указатели на динамически распределяемую память.

Предполагается, что каждый экземпляр объекта ссылается на специально выделенную для него область памяти. Например, класс `String` содержит указатель, ссылающийся на область динамически распределяемой памяти, которая имеет связанные с конкретным объектом `String` символы.

Когда элементы данных одного объекта копируются в элементы данных другого, соответствующие указатели обоих объектов будут иметь одно содержимое, т. е. ссылаться на одну область динамически распределяемой памяти. Эти объекты могут прекратить свое существование в разное время. Например, значение формального параметра функции в листинге 11.3 исчезает, когда функция завершает работу, а фактический аргумент продолжает существовать в пространстве клиента, в функции `main()`. Когда объект исчезает, деструктор освобождает память, на которую ссылаются указатели объекта. Второй существующий объект также теряет свои данные в динамически распределяемой области. Применение объекта с такими данными будет некорректным и даст "ошибку".

Если возвращенная динамически распределяемая память не будет занята немедленно, то такой "фантомный" объект может вести себя совершенно нормально, как если бы его память существовала. Тестирование может показать программисту, что программа корректна.

Когда исчезает второй объект, вызывается деструктор. Обратите внимание, не "снова вызывается". Ранее деструктор вызывался для другого объекта (формального параметра), и он уже уничтожен. Теперь он используется для второго объекта (фактического параметра) и пытается освободить тот же сегмент динамически распределяемой памяти. В C++ такое действие приведет к ошибке. Поведение программы будет не определено, т. е. она будет делать то, что заблагорассудится.

Решение проблем целостности

Есть ряд способов, которыми можно воспользоваться, чтобы избежать неприятностей при передаче как значений параметров-объектов с динамически распределяемой памятью.

Один из них заключается в том, чтобы устранить деструктор, возвращающий динамически распределяемую память системе. Это решение нельзя считать правильным. Им можно воспользоваться в случае, когда программа аварийно завершает работу и нужно выполнить ее для отладки. Отключение деструктора позволит выполнить программу до конца.

Еще один способ состоит в использовании внутри объектов массивов фиксированного размера, а не динамически распределяемой памяти. Такое решение может подойти при частом изменении размера массива. Оно особенно хорошо для программ, работающих с относительно большим числом объектов, и когда отсечение данных, не помещающихся в область фиксированного размера, считается приемлемым с точки зрения целостности приложения.

Лучший способ для передачи параметра — это передача по ссылке, а не по значению. В этом случае устраняются проблемы, создаваемые копированием объектов, ускоряется выполнение программы, исключается необходимость создания и уничтожения временных объектов, вызова конструкторов и деструкторов.

К сожалению, данное решение не универсально. Есть случаи, когда копирование одного объекта в другой не имеет отношения к передаче параметров, и такое решение неприменимо (например, один объект класса инициализируется другим объектом того же класса). Рассмотрим следующий фрагмент программы, где параметр передается функции `operator+=()` по ссылке.

```
String u("Это тест."), v("Все нормально.");
cout << " u = " << u.show() << endl;          // результат OK
cout << " v = " << v.show() << endl;          // результат OK
u += v;                                       // u.operator+=(v); по ссылке
cout << " u = " << u.show() << endl;          // результат OK
cout << " v = " << v.show() << endl;          // OK: передача по ссылке
v.modify("Давайте надеяться на лучшее.");     // нет порчи содержимого памяти
String t = v;                                 // инициализация объекта
cout << " t = " << t.show() << endl;          // OK: корректный результат
t.modify("Все нормально.");                   // изменяются t и v
cout << " t = " << t.show() << endl;          // OK: результат
cout << " v = " << v.show() << endl;          // v также изменен
```

```
u = Это тест.
v = Все нормально.
u = Это тест. Все нормально.
v = Все нормально.
t = Давайте надеяться

t = Все нормально.
v = Все нормально.
```

Рис. 11.12. *Ожидаемый результат выполнения приведенного фрагмента кода*

Этот фрагмент создает два объекта `String` (`u` и `v`), инициализирует их с помощью конструктора преобразования и конкатенирует. Так как объект-аргумент `v` передается функции `operator+=()` по ссылке, порчи содержимого памяти здесь не будет. Объект `v` поддерживает собственную память. При модификации объекта `v` изменяется только этот объект, а не объект `u`. Далее создается еще один объект типа `String`, объект `t`. Ему присваивается текущее состояние объекта `v`. При изменении содержимого объекта `t` объект `v` останется без изменений. На рис. 11.12 показаны предполагаемые результаты выполнения данного фрагмента программы.

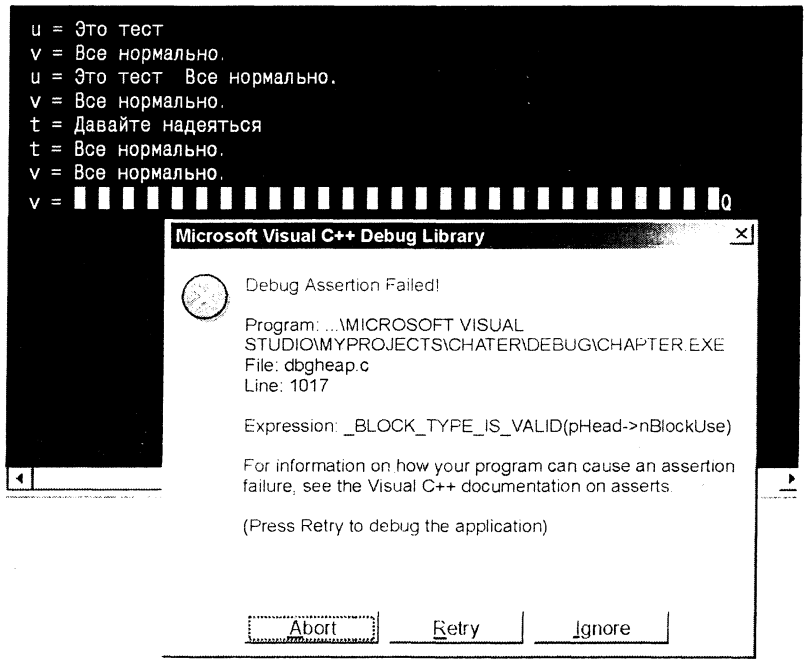


Рис. 11.13. *Результат выполнения программы из листинга 11.4*

Но в реальной жизни бывают исключения. Листинг 11.4 показывает программный код для класса `String` (с параметром, передаваемым перегруженной операторной функции `operator+=()` по ссылке), и клиента, реализующего приведенный выше фрагмент. Фрагмент программы изменен таким образом, чтобы объект `t` создавался во вложенном цикле. Когда вложенный цикл завершает работу и объект `t` исчезает, можно проверить состояние объекта и его целостность. На рис. 11.13 показаны результаты выполнения программы из листинга 11.4.

Листинг 11.4. Инициализация одного объекта с помощью данных другого объекта

```
#include <iostream>
using namespace std;

class String {
  char *str;                          // динамически распределяемый символьный массив
  int len;
public:
  String (int length=0);              // конструктор преобразования по умолчанию
  String(const char*);                // конструктор преобразования
  ~String ();                         // освобождение памяти
  void operator += (const String);    // конкатенация с другим объектом
  void modify(const char*);           // изменение содержимого массива
  const char* show() const;           // возврат указателя массива
  } ;

String::String(int length)
{ len = length;
  str = new char[len+1];
  if (str==NULL) exit(1);
  str[0] = 0; }                       // пустая строка нулевой длины - ОК
```

```
String::String(const char* s)
{  len = strlen(s);                    // определение длины входного текста
   str = new char[len+1];              // выделение достаточной памяти в динамической области
   if (str==NULL) exit(1);             // проверка на успех
   strcpy(str,s); }                    // копировать входящий текст в память

String::~String()
{ delete str; }                        // возврат памяти в динамической области (не указателя)

voidString::operator += (const String& s)      // передача по значению
{  len = strlen(str) + strlen(s.str);  // защита от переполнения
   char* p = new char[len + 1];        // выделение достаточного объема памяти
   if (p==NULL) exit(1);               // проверка на успех
   strcpy(p,str);                      // копирование первой части результата
   strcat (p,s.str);                   // добавление второй части результата
   delete str;                         // важный шаг
   str = p; }                          // теперь p может исчезнуть

const char* String::show() const       // защита данных от изменений
{ return str; }

void String::modify(const char a[])    // здесь нет управления памятью
{  strncpy(str,a,len-1);               // защита от переполнения
   str[len-1] = 0; }                   // правильное завершение строки

int main()
{  cout << endl << endl;
   String u("Проверка");
   String v("Ничего плохого не случится");
   cout << " u = " << u.show() << endl;          // результат OK
   cout << " v = " << v.show() << endl;          // результат OK
   u += v;                                       // u.operator+=(s);
   cout << " u = " << u.show() << endl;          // результат OK
   cout << " v = " << v.show() << endl;          // результат не OK
   v.modify("Давайте надеяться на лучшее.");     // порча содержимого памяти
   {  String t = v;                              // инициализация
      cout << " t = " << t.show() << endl;       // OK, корректный результат
      t.modify("Ничего плохого не случится.")    // изменяется t и v
      cout << " v = " << v.show() << endl; }     // v также изменился
   cout << " v = " << v.show() << endl;          // t больше нет, v теряет память
   return 0;
   }
```

При создании строкового объекта t типа String (для него отводится память в стеке, так как t — локальная автоматическая переменная) выделенной памяти достаточно для символьного указателя и целого. Вызывается конструктор. В клиенте можно видеть символ присваивания, но это инициализация. Важно знать, какой вызывается конструктор после создания объекта. Ответ зависит от данных, подставляемых клиентом при создании объекта. В листинге 11.4 функция main() подставляет один фактический аргумент — существующий объект v. Следовательно, это должен быть конструктор с одним параметром того же типа, в данном случае конструктор класса String.

Каково же имя данного конструктора с одним параметром? Как говорилось в главе 9, это конструктор копирования, поскольку он копирует данные из одного объекта в другой. Между тем класс String не имеет конструктора копирования. Это не означает, что попытка вызова отсутствующего конструктора даст синтаксическую ошибку. Компилятор генерирует вызов конструктора копирования,

подставляемого системой. Конструктор копирует поля аргумента-объекта в поля создаваемого объекта. Для класса String подставляемый системой конструктор выглядит следующим образом:

```
String::String(const String& s)   // конструктор, подставляемый системой
{ len = s.len;                    // копирование длины текстового объекта
  str = s.str; }                  // копирование указателя текстового объекта
```

На рис. 11.14 показана работа конструктора. При создании объекта t типа String его полю len присваивается значение 9, а поле str устанавливается на ту же область динамически распределяемой памяти, на которую указывает поле str объекта v.

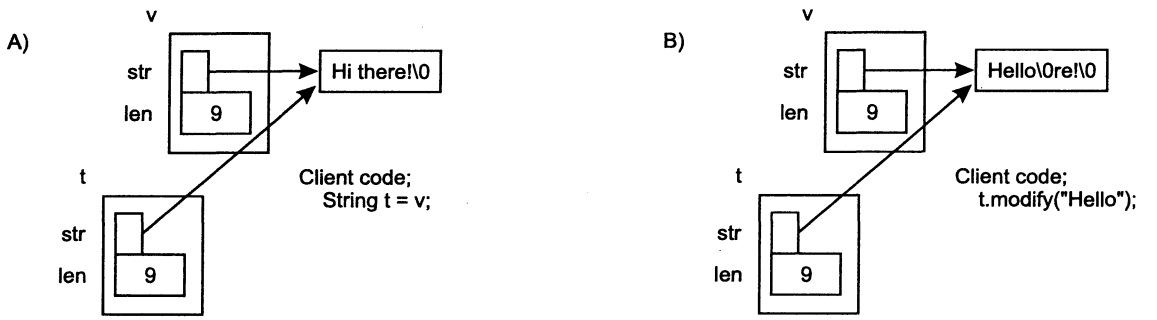


Рис. 11.14. *Диаграмма памяти для инициализации одного объекта String данными другого объекта*

Объекты t и v ссылаются на один сегмент динамически распределяемой памяти. Ранее он был выделен для объекта v, но теперь на него ссылается еще и объект t. Каждый объект считает, что эта область памяти принадлежит только ему. Еще хуже, что здесь имеет место передача по значению. При передаче по значению фактический аргумент существует в области действия клиента, а формальный параметр — в области сервера. При выполнении доступен только один объект. Оба объекта существуют в области действия клиента. Обращение к ним и модификация происходят в одной и той же области.

Объекты используют одну и ту же область динамически распределяемой памяти. Для клиента они являются синонимами. Если в клиенте изменяется объект t, меняется и объект v.

Если подумать, то странным это кажется только с точки зрения общепринятого подхода в программировании. При пояснении классов уже рассказывалось, что у студентов часто возникают трудности с простым исходным кодом с целыми числами:

```
int v = 10;  int t = v;  t = 20;                    // чему теперь равно v?
```

Большинство программистов считают, что после изменения t переменная v остается прежней, так как t и v занимают разные области памяти. Другие полагают, что переменные v и t — одно и то же, так что не удивительно, когда при изменении t меняется и значение v.

Если переменные являются синонимами, то изменение одной из них отражается на другой. Помните, что часто одна переменная является обычной переменной, а другая — ссылкой.

```
int v = 10;  int& t = v;  t = 20;                   // чему теперь равно v?
```

В данном примере трудно разобраться. В записи утверждается, что две переменные v и t одно и то же. Не удивительно, что v изменяется после изменения t. Теперь v равно 20. Это должны усвоить программисты, применяющие C++.

Семантика копирования и семантика значений

Существуют два общих подхода в программировании, соответствующих двум разным концепциям вычислительной техники — *семантика значения* и *семантика ссылок*. (Под семантикой здесь понимается смысл копирования данных.)

В самом распространенном подходе в программировании используется семантика значений. Каждый вычислительный объект (например, переменная встроенного типа или объект определяемого программистом типа) имеет собственную отдельную область памяти. Если один вычислительный объект приравнять к другому, повторится битовая последовательность одного объекта в памяти другого объекта. В C++ семантика значений используется как для встроенных переменных, так и для объектов определяемого программистом типа.

```
int v = 10;  int t = v;  t = 20;                    // семантика значений, v = 10
```

Такая интуиция более распространена по следующей причине: с ее точки зрения, когда объекты имеют одно значение, они используют две разные битовые последовательности, и изменение одного объекта не влияет на последовательность битов, уже существующую в другом объекте.

В другой, менее распространенной интуиции программирования используется семантика ссылок. Когда вычислительному объекту присваивается значение, он получает ссылку (или указатель) на это значение. Приравнивание вычислительных объектов означает присваивание их ссылкам одной и той же области памяти. Когда изменяется массив символов, на который указывает один из этих объектов, меняется другой объект, так как оба объекта ссылаются на одну область. В C++ такая семантика ссылок используется для указателей и ссылок, при передаче параметров по ссылке или по указателю, для массивов и связанных структур данных с указателями.

```
int v = 10;  int& t = v;  t = 20;                   // согласно семантике ссылок, v = 20
```

Семантика ссылок распространена меньше. Она используется в основном из соображений производительности (например, исключает копирование объектов при передаче параметров). Иногда она действует неявно, как в данном примере. Что касается C++, то программист всегда должен помнить о разнице между семантикой значений и семантикой ссылок.

Вы узнали еще не обо всех неприятностях с программой из листинга 11.4. Когда при ее выполнении достигается закрывающая фигурная скобка вложенного цикла, объект `t` должен исчезнуть, так как он определен в этом вложенном цикле. Объект `v` определен во внешней области действия функции `main()` и должен быть доступен для дальнейшего использования. В листинге 11.4 показана попытка вывода значения `v` в конце функции `main()`. Обратите внимание, что этот оператор отделен от предшествующего оператора вывода только закрывающей фигурной скобкой вложенной области действия. На первый взгляд, между двумя операторами в клиенте не происходит никаких событий. Следовательно, они должны давать один и тот же результат. Но результат разный. В этом случае рекомендуется выработать вам собственный подход, который поможет читать подобные фрагменты исходного кода.

Первый оператор дает вполне нормальный результат (см. рис. 11.14). Это не совсем то, что можно было бы ожидать, но, по крайней мере, он есть. Второй результат — просто "мусор". Что произошло между двумя операторами? Когда достигается закрывающая фигурная скобка области действия, для локального объекта `t` в этой вложенной области вызывается деструктор класса `String`. Как видно из листинга 11.4 и из рис. 11.14, данный деструктор освобождает динамически распределяемую память, принадлежащую объекту `b`, но система этого не помнит. Она запоминает лишь то, что память, на которую ссылается `str`, следует освободить согласно деструктору класса `String`. У объекта `v` уже нет динамической памяти, но никто об этом не знает. Формально он находится в области действия. Но это только на первый взгляд.

Ситуация аналогична передаче параметра по значению, но это еще не конец. Когда программа достигает закрывающей фигурной скобки, объект v должен исчезнуть согласно правилам области действия. Перед этим вызывается деструктор и пытается освободить уже освобожденную динамически распределяемую область памяти. Программа некорректна. Она вышла из-под контроля.

Конструктор копирования, определяемый программистом

Конструктор копирования должен распределять динамическую память для целевого объекта в соответствии с операцией конкатенации, о которой рассказывалось в предыдущем разделе. Вот его алгоритм:

1. Скопировать длину символьного массива параметра в поле `len` целевого объекта.
2. Выделить динамически распределяемую память: установить на нее указатель `str` целевого объекта.
3. Проверить, успешно ли выделена память. Отказать, если в системе нет памяти.
4. Скопировать символы из целевого объекта в только что выделенную память.

Конструктор копирования, определяемый программистом и позволяющий решить проблему:

```
String::String(const String& s)      // определяемый программистом
                                     // конструктор копирования
{ len = s.len;                       // длина исходного текста
  str = new char[len+1];             // запрос отдельной динамической памяти
  if (str == NULL) exit(1);          // проверка на успех
  strcpy(str, s.str); }              // копирование исходного текста
```

Обратите внимание, что параметр s передается по ссылке. Это ссылка на фактический аргумент-объект. При передаче параметра не происходит копирования элементов данных аргумента. Динамическая память фактического аргумента-объекта копируется в динамическую память целевого объекта.

Это менее эффективно, чем элементное копирование, показанное в листинге 11.4. Семантика значений работает медленнее, чем семантика ссылок. Здесь вы имеете дело со значениями, а не со ссылками или указателями. Между тем, семантика значений надежна. Вспомните вариант клиента, который привел ко всем этим проблемам.

```
String t = v;        // нет проблем, если используется конструктор копирования
```

После выполнения данной строки указатели `str` в объектах `v` и `t` ссылаются на разные области динамически распределяемой памяти. Проблема целостности решена.



Внимание Если среди элементов данных класса встречаются указатели и объекты этого класса работают с динамически распределяемой памятью, то разработчик класса должен решить, что именно нужно в нем использовать — семантику ссылок или семантику значений. Если необходима семантика значений и требуется инициализировать один объект значением другого объекта, убедитесь, что этот класс имеет определяемый программистом конструктор копирования.

```
u = Это тест.
v = Все нормально.
u = Это тест. Все нормально.
v = Все нормально.
t = Давайте надеяться
t = Все нормально.
v = Давайте надеяться
v = Давайте надеяться
```

Рис. 11.15.
Результат выполнения программы из листинга 11.5

Листинг 11.5 показывает программу из листинга 11.4, где класс String определяет свой собственный конструктор, поддерживающий для инициализации объекта семантику значений. Результат программы представлен на рис. 11.15. Как видно, проблема целостности исчезла. Объекты t и v типа String больше не являются синонимами. Когда объект t изменяется, объект v остается тем же. При завершении вложенной области действия и исчезновении объекта t объект v может использоваться в клиенте. Трассировка кода и его результата доказывает существование связи между двумя объектами.

Листинг 11.5. Использование конструктора копирования для инициализации одного объекта с помощью данных другого объекта

```
#include <iostream>
using namespace std;

class String {
  char *str;                              // динамически распределяемый символьный массив
  int len;
  char* allocate(const char* s)           // закрытая функция
  {  char *p = new char[len+1];           // выделение динамической памяти для объекта
     if (p==NULL) exit(1);                // проверка на успех; выход в случае неудачи
     strcpy(p,s);                         // копирование текста в динамическую память
     return p; }                          // возврат указателя на динамическую память
public:
  String (int length=0);                  // конструктор преобразования/по умолчанию
  String(const char*);                    // конструктор преобразования
  String(const String& s);                // конструктор копирования
  ~String ();                             // освобождение динамической памяти
  void operator += (const String&);       // конкатенация с другим объектом
  void modify(const char*);               // изменение содержимого массива
  const char* show() const;               // возврат указателя массива
  } ;

String::String(int length)
{ len = length;
str = allocate(""); }                     // копирование пустой строки в динамическую память

String::String (const char* s)
{  len = strlen (s);                      // определение данных исходного текста
   str = allocate(s.); }                  // выделение памяти, копирование входящего текста

String::String(const String& s)           // конструктор копирования
{  len = s.len;                           // определение длины исходного текста
   str = allocate (str); }                // выделение памяти, копирование входящего текста

String::~String()
{ delete str; }                           // возврат динамической памяти (не указателя)

void String::operator += (const String& s)        // параметр-ссылка
{  len = strlen(str) + strlen(s.str);     // защита от переполнения
   char* p = new char[len + 1];           // выделение достаточного объема памяти
   if (p==NULL) exit(1);                  // проверка на успех
   strcpy(p,str);                         // копирование первой части результата
   strcat (p,s.str);                      // добавление второй части результата
   delete str;                            // важный шаг
   str = p; }                             // теперь p может исчезнуть
```

```
const char* String::show() const                    // защита данных от изменений
{ return str; }

void String::modify(const char a[])                 // передача по значению
{  strncpy(str,a,len-1);                            // защита от переполнения
   str[len-1] = 0; }                                // правильное завершение строки

int main()
{  cout << endl << endl;
   String u("Проверка");
   String v("Ничего плохого не случится");
   cout << " u = " << u.show() << endl;             // результат OK
   cout << " v = " << v.show() << endl;             // результат OK
   u += v;                                          // u.operator+=(v);
   cout << " u = " << u.show() << endl;             // результат OK
   cout << " v = " << v.show() << endl;             // OK - передача по ссылке
   v.modify("Давайте надеяться на лучшее.");        // порча содержимого памяти
   {  String t = v;
      t.modify("Ничего плохого не случится")        // меняем только t
      cout << " t = " << t.show() << endl;          // OK, корректный результат
      cout << " v = " << v.show() << endl; }        // v также изменился
   cout << " v = " << v.show() << endl;             // t больше нет, v теряет память
   return 0;
   }
```

В Листинге 11.5 класс String имеет три конструктора. Они выделяют память в динамически распределяемой области и инициализируют ее содержимое. В первом конструкторе преобразования инициализирующие данные — это пустая строка, завершаемая нулем. Во втором конструкторе преобразования это символьный массив, подставляемый клиентом в фактическом аргументе. В конструкторе копирования инициализирующими данными является символьный массив внутри объекта, поставляемого клиентом. Поскольку этот массив находится в динамической памяти, он не имеет имени, а ссылка на него осуществляется через указатель str. Так как параметр-объект s принадлежит к тому же классу String, что и целевой инициализируемый объект, конструктор копирования имеет право доступа к этому закрытому указателю str с помощью уточненного имени s.str.

Вполне естественно, что разные конструкторы используют аналогичные алгоритмы, так как вид полученного в результате объекта не должен зависеть от вызванного при его создании конструктора. Если класс содержит один или два конструктора, повторите код. Если алгоритм используется очень часто, то программисты обычно инкапсулируют его в частную функцию и вызывают ее из разных компонентных функций. Данная функция должна быть закрытой, так как клиент не заинтересован в прямой работе с памятью объекта. Это детали нижнего уровня, которые не должны запутывать алгоритм клиента и занимающегося им программиста. Подобная закрытая функция представлена в листинге 11.5. Когда она копирует свой параметр в выделенную динамическую память, то использует имя указателя p.

```
char* allocate(const char* s)    // закрытая функция
{ char *p = new char[len+1];     // выделение памяти для объекта
  if (p==NULL) exit(1);          // проверка на успех; выход, если не повезло
  strcpy(p,s);                   // копирование текста в динамическую память
  return p; }                    // возврат указателя на динамическую память
```

Листинг 11.5 показывает, что первый конструктор преобразования передает функции allocate() пустую строку, второй отправляет свой собственный параметр — символьный массив, а конструктор копирования передает allocate() символьный массив своего параметра, т. е. s.str.

Когда один объект инициализирует другой объект, вызывается конструктор копирования. Это неизбежно. Вопрос в том, какой именно конструктор вызывается. Если класс не предусматривает свой собственный конструктор копирования, то компилятор генерирует вызов системного конструктора, который копирует элементы данных объекта. Если для объектов этого класса не выделяется динамически распределяемая память, то все замечательно. Если объекты используют индивидуальные сегменты динамически распределяемой памяти (семантику значений), то применение предусмотренного системой конструктора копирования подрывает целостность приложения. Чтобы сохранить целостность программы, в классе следует использовать собственный конструктор копирования, который выделяет целевому объекту свою динамически распределяемую память.

В предыдущем предложении "следует предусмотреть" подчеркивает взаимосвязь "клиент-сервер" между разными сегментами программы C++ и между разными уровнями понимания. С помощью клиентской программы обрабатываются объекты. Сервер поддерживает клиента, реализуя вызываемые клиентом функции-члены. Конструкторы вызываются неявно, но при этом не изменяется соотношение клиент-сервер.

Если в приложении необходима семантика копирования, в классах с динамическим управлением памятью используются конструкторы копирования для других контекстов, когда один объект инициализирует другой. Одним из таких контекстов является передача параметров-объектов по значению. При наличии соответствующего конструктора копирования будет замечательно работать первая версия перегруженной операции конкатенации из листинга 11.3.

```
void String::operator += (const String s)        // параметр-объект
{ len = strlen(str) + strlen(s.str);             // общая длина
  char *p = new char[len + 1];          // распределение динамической памяти
  if (p==NULL) exit(1);                 // проверка на успех
  strcpy(p,str);                        // копирование первой части результата
  strcat (p,s.str);                     // добавление второй части результата
  delete str;                           // важный шаг
  str = p; }                            // str указывает на новую память
```

При вызове данной функции и создании фактического аргумента вызывается определенный программистом конструктор копирования. Он выделяет динамическую память для формального параметра s. Когда эта функция завершает работу и для формального параметра вызывается деструктор, освобождается его собственная динамическая память, а не динамическая память, принадлежащая фактическому аргументу. Проблема, связанная с целостностью, исчезает. Однако сохраняется проблема производительности. Когда параметр передается по значению, вызов операции конкатенации предусматривает создание объекта, вызов конструкторов копирования, выделение динамически распределяемой памяти, копирование символов из одного объекта в другой, вызов деструктора и освобождение динамической памяти. Для вызова по ссылке не требуется проведение подобных операций. Семантика ссылок решает проблему производительности, поскольку устраняется излишнее копирование.



Осторожно! Не передавайте объекты функциям по значению. Если объекты имеют внутренние указатели и работают с динамически распределяемой памятью, не передавайте эти объекты по значению. Если же необходимо отправить такие объекты по значению, определите конструктор копирования.

Возврат по значению

При возврате объекта из функции по значению применяйте семантику значений. В главе 10 уже обсуждались классы, не работающие с динамически распределяемой памятью. Поскольку ситуация с возвратом объекта из функции в точности такая же, как и при инициализации одного объекта другим, рассмотрим ее.

```
Исходный: ''
Исходный: ''
Исходный: ''
Исходный: ''
Создан: 'Атланта'
Создан: 'Бостон'
Создан: 'Чикаго'
Создан: 'Денвер'
Введите название города для поиска: Бостон
Создан: 'Бостон'
Город Бостон найден
```

Рис. 11.16. *Результат выполнения программы из листинга 11.6*

Листинг 11.6 представляет еще одну версию класса `String`. В каждый конструктор включены отладочные операторы и добавлена перегруженная операция сравнения, реализованная как функция-член. Кроме того, здесь добавлена функция клиента `enterData()` и обновлена функция `main()`. Программа просит пользователя ввести название города и ищет это имя в базе данных. Для простоты база данных определена в функции `main()` как массив символов, и используется простой последовательный поиск. Результаты выполнения показаны на рис. 11.16.

Листинг 11.6. Использование конструктора копирования для возврата объекта из функции

```
#include <iostream>
using namespace std;

class String {
  char *str;                              // динамически распределяемый символьный массив
  int len;
  char* allocate(const char* s)           // закрытая функция
  {  char *p = new char[len+1];           // выделение динамической памяти для объекта
     if (p==NULL) exit(1);                // проверка на успех; выход в случае неудачи
     strcpy(p,s);                         // копирование текста в динамическую память
     return p; }                          // возврат указателя на динамическую память
public:
  String (int length=0);                  // конструктор преобразования/по умолчанию
  String(const char*);                    // конструктор преобразования
  String(const String& s);                // конструктор копирования
  ~String ();                             // освобождение динамической памяти
  void operator += (const String&);       // конкатенация с другим объектом
  void modify(const char*);               // изменение содержимого массива
  bool operator == (const String&) const;          // сравнение содержимого
  const char* show() const;               // возврат указателя массива
} ;

String::String(int length)
{  len = length;
  str = allocate("");                     // копирование пустой строки в динамическую память
  cout << " Исходный: '" << str << "'\n"; }

String::String(const char* s)
{  len = strlen(s);                       // определение длины исходного текста
  str = allocate(s);                      // выделение памяти, копирование текста
  cout << " Созданный: '" << str << "'\n"; }

String::String(const String& s)           // конструктор копирования
{  len = s.len;                           // определение длины исходного текста
  str = allocate(s.str);                  // выделение памяти, копирование текста
  cout << " Скопированный: '" << str << "'\n"; }
```

```
String::~String()
{ delete str; }                                  // возврат динамической памяти (не указателя)

void String::operator += (const String& s)       // параметр-ссылка
{  len = strlen(str) + strlen(s.str);            // защита от переполнения
   char* p = new char[len + 1];                  // выделение достаточного объема памяти
   if (p==NULL) exit(1);                         // проверка на успех
   strcpy(p,str);                                // копирование первой части результата
   strcat (p,s.str);                             // добавление второй части результата
   delete str;                                   // важный шаг
   str = p; }                                    // теперь p может исчезнуть

bool String::operator==(const String& s) const           // сравнение содержимого
{ return strcmp(str,s.str)==0: }                 // при совпадении strcmp возвращает 0

const char* String::show() const                 // защита данных от изменений
{ return str; }

void String::modify(const char a[])              // передача по значению
{  strcpy(str,a,len-1);                          // защита от переполнения
   str[len-1] = 0; }

String enterData()
{  cout << "Введите название города для поиска: ";          // запрос пользователю
   char data[200];                               // грубое решение
   cin >> data;                                  // принять ввод от пользователя
   return String(data); }                        // вызов конструктора

int main()
{  enum { MAX = 4;
   String data[4];                               // база данных объекта
   char *c[4] = { "Атланта", "Бостон", "Чикаго", "Денвер" };
   for (int j=0; j<MAX; j++)
   { data[j] += c[j]; }                          // data[j].operator+=(c[j]);
   String u = enterData();                       // аварийно завершается без конструктора
                                                 // копирования
   int i;
   for (i=0; i < MAX; i++)                       // i определено вне цикла
     { if (data[i] == u) break; }                // выход, если строка найдена
  if (i == MAX)
     cout << "Город << u.show() << " не найден\n";
   else
     cout << " Город << u.show() << " найден\n";
   return 0;
   }
```

Когда в функции `main()` создается массив объектов, для каждого из них вызывается заданный по умолчанию конструктор `String` (например, первый конструктор преобразования со значением по умолчанию). Конструктор выделяет память для пустой строки нулевой длины и выводит сообщение "Исходный". Когда вызывается функция `operator+=()` и к содержимому каждого объекта добавляются названия городов, массив символов передается операции сравнения как параметр. Перегруженная операторная функция ожидает получения параметра `String`. Следовательно, вызывается второй конструктор преобразования, который для каждого элемента массива выводит сообщение "Создан".

Вызывается функция enterData(). Она запрашивает у пользователя название города и передает его как аргумент конструктору преобразования String. На экране появляется выводимое конструктором сообщение "Создан". Поскольку объект u в функции main() создается только при вызове enterData(), вызываемый в enterData() конструктор используется как конструктор для объекта u в функции main(). Конструктор копирования не вызывается. Объекты String работают с динамической памятью, однако целостность программы сохраняется. Конструктор преобразования выделяет для объекта u в функции main() индивидуальную память в динамически распределяемой области.

Измените функцию enterData(). Добавьте еще один локальный объект для хранения данных пользователей.

```
String enterData()
{ cout << "Введите название города для поиска: ";      // запрос пользователю
  char data[200];                                      // грубое решение
  cin >> data;                                         // принять ввод от пользователя
  String x = data;                                     // конструктор преобразования
  return x; }                                          // конструктор копирования
```

Изменения незначительны. Если бы x была переменной встроенного типа, то все осталось бы по-прежнему. Для объектов с динамически распределяемой памятью все иначе. При создании объекта x вызывается конструктор преобразования. Между тем, когда функция завершает работу, объект u в функции main() инициализируется с помощью конструктора копирования. Если определяемый программистом конструктор копирования не реализован, то используется системный конструктор копирования. Он копирует элементы данных объекта x в элементы данных объекта u и не выделяет динамическую память. Указатели str объектов u и x ссылаются на одну область динамически распределяемой памяти. Когда функция enterData() завершает работу и объект x исчезает, вызывается деструктор String. Он освобождает динамическую область памяти, на которую ссылается указатель str в объекте x. Динамическая память объекта u освобождается при создании объекта.

```
Исходный: ''
Исходный: ''
Исходный: ''
Исходный: ''
Создан: 'Атланта'
Создан: 'Бостон'
Создан: 'Чикаго'
Создан: 'Денвер'
Введите название города для поиска: Москва
Создан: 'Москва'
Скопирован: 'Москва'
Город Москва найден
```

Рис. 11.17. *Результат выполнения программы из листинга 11.6 с модифицированной функцией enterData() и конструктором копирования*

В результате такого подхода программа аварийно завершает работу. Однако она уже некорректна. Нужен определяемый программистом конструктор копирования.

Все работает нормально, когда подставляется определяемый программистом конструктор копирования. Пример результата выполнения программы представлен на рис. 11.17.

Вывод показывает, что после ввода строки пользователем для локального объекта x в функции main() вызывается конструктор преобразования, затем конструктор копирования для локального объекта u в функции main(). Данная версия работает медленнее предыдущей. Однако важно то, что она ведет себя не так, как предыдущая. И еще важнее, что, если бы переменные x и u были встроенного типа, то такие изменения не повлияли бы на поведение программы. Подобные эксперименты со встроенными типами помогают программисту найти свое решение. Несмотря на все усилия, встроенные и определяемые программистом типы интерпретируются в C++ по-разному. Работа с объектами требует изменения интуиции программиста. Вот почему важно проследить за последовательностью событий. Нужно уметь связывать структуры клиента с неявно вызываемыми функциями класса.

Ограничения для эффективности конструктора копирования

Хотелось бы внести в программу еще одно небольшое изменение, на этот раз в клиенте. Вместо определения объекта u в функции main() и немедленной его инициализации после определения этого объекта (с помощью конструктора по умолчанию) присвоим ему то, что вводит пользователь при вызове функции enterData().

```
int main()
{ enum { MAX = 4) ;

  // создание базы данных с названиями городов
  // String u = enterData();              // аварийно завершается без
                                          // конструктора копирования
  String u;                               // конструктор по умолчанию
  u = enterData();                        // аварийно завершается: конструктор
                                          // копирования не поможет

  // поиск города, вывод результатов
  return 0;
  }
```

После внесения изменений моя система аварийно завершает работу. Не стоит приводить еще одно диалоговое окно с бесполезной информацией об источнике проблемы. Кроме того, это был бы пример выполнения на конкретной машине под конкретной ОС. Важно то, что программа некорректна. Хотя она правильно компилируется, она не должна выполняться. Так как компилятор не сообщает о некорректности программы, именно интуиция программиста должна подсказать ему, что происходит в ее недрах.

Перегрузка операции присваивания

В некоторых случаях инициализация и присваивание объекта в C++ различаются. Если приходится иметь дело со встроенными типами данных, такое различие часто бывает академическим. Рассмотрим пример:

```
int v = 5; int u = v;                  // переменная инициализируется
```

Сравните его с таким примером:

```
int v = 5; int u; u = v;               // присваивание переменной u
```

В первом примере переменная u инициализируется при определении. Во втором происходит присваивание переменной u после ее определения. Для встроенных переменных конечный результат будет одним и тем же, но когда эти вычислительные объекты представляют собой объекты определяемых программистом типов, работающих со своей собственной памятью, разница важна.

```
String v = "Hello"; String u = v;          // объект u инициализируется
String v = "Hello"; String u; u = v;       // объект u присваивается
```

Если в классе нет конструктора копирования, первая строка может быть для вас неожиданной. Вторая строка приведет к проблемам при отсутствии в классе совмещенной операции присваивания. Конструктор копирования для второй строки не вызывается.

Проблемы системной операции присваивания

Если класс имеет совмещенную операцию присваивания, она вызывается во второй строке приведенного выше примера. В противном случае компилятор использует собственную операцию присваивания. Она похожа на конструктор копирования и дублирует поля данных объекта с правой стороны присваивания в поля компонентов данных в левой части операции присваивания.

Подобно подставляемому системой конструктору копирования, эта предусмотренная системой операция присваивания всегда доступна. Для классов без динамического управления памятью (например, `Complex`, `Rational`, `Rectangle`) такая "системная" операция присваивания вполне подходит. Для классов, динамически управляющих своей памятью, данная подставляемая системой операция присваивания вызывает проблемы.

Когда присваивание выполняется для объектов `String`, копируются отдельные элементы данных объектов. Указатель `str` объекта в левой части присваивания ссылается на то же место в динамически распределяемой памяти, что и указатель `str` объекта в правой части операции присваивания. Объекты становятся синонимами. Если изменить один объект, например `u`, то меняется и другой объект, в данном случае `v`.

Когда объект, например `u`, уничтожается операцией `delete`, для него вызывается деструктор. Освобождается память, на которую ссылается указатель `str`. В результате другой объект (в данном случае `v`) теряет свою динамическую память, хотя в программе он еще существует. Любое использование такого объекта будет некорректным. Когда этот объект уничтожается, для него вызывается деструктор. Он пытается освободить память, на которую ссылается указатель `str`. Однако эта память уже освобождена. Как пояснялось ранее, попытка повторного освобождения динамически распределяемой памяти приводит к непредсказуемому поведению программы. Такие действия семантически некорректны, хотя синтаксически правильны.

Поиск причины проблемы будет затруднительным, так как трудности не связаны с результатами выполнения программы. Конструктор копирования не позволяет от нее избавиться, поскольку во время операции присваивания он не вызывается. В C++ присваивание и инициализация — разные понятия.

Перегруженная операция присваивания: первая версия (утечка памяти)

Решение данной проблемы заключается в реализации для класса перегруженной операции присваивания. Она гарантирует, что объекты в левой и в правой частях присваивания не будут ссылаться на одну и ту же область в динамически распределяемой памяти.

Встроенная операция присваивания в C++ — двухместная операция (с двумя операндами). Она содержит операнды в левой и в правой частях. Это же относится к перегруженной операции присваивания, определяемой программистом. Следовательно, интерфейс такой операции аналогичен интерфейсу конструктора копирования: объект в левой части операции является получателем сообщения, а объект в правой части — параметром.

```
u = v;                  // u.operator=(v);
```

Это означает, что перегруженная операция присваивания для класса `String` должна иметь следующий интерфейс:

```
void String::operator = (const String& s);          // операция присваивания
```

Операция присваивания должна копировать отличные от указателей элементы данных из объекта-параметра в целевой объект, выделять область памяти достаточного размера и копировать содержимое динамически распределяемой памяти параметра в динамическую память целевого объекта. Эти действия аналогичны действиям конструктора копирования.

1. Копирование длины символьного массива (параметра) в элемент `len` целевого объекта.
2. Выделение динамической памяти и установка на нее указателя `str` целевого объекта.
3. Проверка на успешное распределение памяти. Отказ, если в системе нет памяти.
4. Копирование символов из объекта-параметра в выделенную область памяти.



Внимание Если нужно присвоить один объект другому и объект работает с динамически распределяемой памятью, то следует убедиться, что класс содержит перегруженную операцию присваивания. Конструктора копирования здесь недостаточно.

Ниже приведена версия операции присваивания. Хотя она работает медленнее, чем операция присваивания, предусмотренная системой, но сохраняет семантику значений.

```
void String::operator = (const String& s)
{ len = s.len;                  // копирование данных, отличных от указателя
  str = new char[len + 1];      // выделение собственной области
                                // в динамической памяти
  if (str == NULL) exit(1);     // проверка на успех
  strcpy(str,s.str); }          // копирование данных в динамически
                                // распределяемую область
```

Такая операция присваивания интерпретирует целевой объект точно так же, как конструктор копирования, т. е. как будто это новый объект, не имеющий предыстории. В случае конструктора копирования так оно и есть, однако в случае операции присваивания это не так. Целевой объект u уже был создан ранее, т. е. конструктор вызывался при создании объекта и указатель `str` был установлен на область в динамически распределяемой памяти. Операция присваивания игнорирует эту динамически распределяемую память. Она устанавливает указатель `str` на другую область в динамической памяти, а память, выделенная ранее для объекта, теряется. Следовательно, данная операция присваивания приводит к утечке памяти. Это вторая опасность в программе C++. Первая неприятность связана с повторным освобождением памяти.

Каков же выход? В отличие от конструктора копирования, операция присваивания должна предварительно освобождать ресурсы (память), используемые целевым объектом. Ниже — улучшенная версия перегруженной операции присваивания:

```
void String::operator = (const String& s)
{ delete str;                   // это нельзя сделать в конструкторе копирования
  len = s.len;                  // копирование данных, отличных от указателя
  str = new char[len + 1];      // выделение собственной области
                                // в динамической памяти
  if (str == NULL) exit(1);     // проверка на успех
  strcpy(str,s.str); }          // копирование данных в динамически
                                // распределенную область
```

Перегруженная операция присваивания: следующая версия (самоприсваивание)

Такая операция присваивания вполне адекватна. В большинстве случаев она будет работать нормально. Проблема, связанная с операцией присваивания, состоит в том, что в коде клиента не поддерживается присваивание следующего вида:

```
u = u;      // u.operator = (u); такое приходится делать нечасто, не так ли?
```

Эта бесполезная операция, но в C++ она вполне законна. Компилятор не помечает данный оператор как синтаксическую ошибку. Подобная операция присваивания просто освобождает в первом операторе функции `operator=()` динамическую память объекта-аргумента. При выполнении библиотечной функции `strcpy()` символы из строки, для которой только что выделена память, копируются сами в себя. Результат копирования перекрывающихся областей памяти не будет определен (опять головная боль), но даже если бы он был обозначен, содержимое динамически распределяемой памяти объекта было бы навсегда потеряно.

Как ни странно, подобное "самоприсваивание" — не такая уж редкость. Оно часто встречается в алгоритмах сортировки и в операциях с указателями. Для предотвращения возврата динамически распределяемой памяти объекта операция проверяет, ссылается параметр на тот адрес, где находится целевой объект, или нет. Для этого используется указатель `this`.

```
void String::operator = (const String& s)
{ if (&s == this) return;     // предотвращает потерю памяти
                              // при самоприсваивании
  delete str;                 // это нельзя сделать в конструкторе копирования
  len = s.len;                // копирование данных, отличных от указателя
  str = new char[len + 1];    // выделение собственной области
                              // в динамической памяти
  if (str == NULL) exit(1);   // проверка на успех
  strcpy(str,s.str); }        // копирование данных в динамически
                              // распределяемую область
```

Подобную проверку можно было бы провести и в клиенте перед вызовом операции присваивания. Однако в этом случае обязанности возлагаются на клиента, а не на сервер.

Еще одно решение состоит в проверке установки указателей `str` в целевом объекте и в объекте-параметре на одну и ту же область в динамически распределяемой памяти. В операторной функции данная проверка может выглядеть так:

```
if (str == s.str) return;     // одна и та же динамически распределяемая память?
```

Первый способ для предотвращения проблемы используется очень часто. Возможно потому, что указатель `this` более привлекателен для программистов.

Перегруженная операция присваивания: еще одна версия (цепочка выражений)

Данную перегруженную операцию присваивания следует использовать для всех классов, работающих с динамической памятью. Между тем, это присваивание не поддерживает цепочки выражений, когда в нем используется значение, возвращаемое предыдущей операцией.

```
t = u = v;                    // возвращает тип void - это не поддерживается
```

В клиенте всегда можно указать последовательность двухместных операций:

```
u = v;              // двухместная операция: u.operator=(v);
t = u;              // двухместная операция: t.operator=(u);
```

Между тем снова встает вопрос об одинаковой интерпретации встроенных типов и типов, определяемых программистом. Для переменных встроенных типов такая цепочка в программе C++ вполне законна. Следовательно, она должна допускаться для всех переменных определяемых программистом типов.

Операция присваивания ассоциативна справа налево, поэтому смысл цепочки следующий:

```
t = (u = v);        // возвращает тип void - это не поддерживается
```

Это означает, что операция присваивания должна возвращать значение, подходящее для использования фактического аргумента в другой операции присваивания, т. е. она должна возвращать значение того типа, которому принадлежит операция присваивания.

```
String String::operator = (const String& s)              // возвращает объект
{ if (&s == this) return *this;  // предотвращает потерю памяти
                                 // при самоприсваивании
  delete str;                   // это нельзя сделать в конструкторе копирования
  len = s.len;                   // копирование данных, отличных от указателя
  str = new char[len + 1];       // выделение собственной области
                                 // в динамической памяти
  if (str == NULL) exit(1);      // проверка на успех
  strcpy(str,s.str);             // копирование данных в динамически
                                 // распределяемой области
  return *this; }
```

Листинг 11.7 представляет модифицированную версию программы из листинга 11.6. Здесь добавлена перегруженная операция присваивания. Для запроса динамически распределяемой памяти и проверки ее успешного выделения вызывается закрытая функция `allocate()`. Чтобы сократить объем отладочного вывода, здесь удалены операторы вывода сообщений "Исходный" из используемого по умолчанию конструктора. Кроме того, исключен вызов операции конкатенации `operator+=()` в цикле клиента, загружающий названия городов в базу данных. Она заменена на операцию присваивания. Результат программы показан на рис. 11.18.

```
Создан: 'Атланта'
Присвоен: 'Атланта'
Скопирован: 'Атланта'
Создан: 'Бостон'
Присвоен: 'Бостон'
Скопирован: 'Бостон'
Создан: 'Чикаго'
Присвоен: 'Чикаго'
Скопирован: 'Чикаго'
Создан: 'Денвер'
Присвоен: 'Денвер'
Скопирован: 'Денвер'
Введите название города для поиска: Денвер
Создан: 'Денвер'
Присвоен: 'Денвер'
Скопирован: 'Денвер'
Город Денвер найден
```

Рис. 11.18. *Результат выполнения программы из листинга 11.7*

Как можно видеть, проблема целостности исчезла. С объектами типа `String` можно работать точно так же, как с объектами встроенных числовых типов. Допускается их создание без инициализации, инициализация из символьного массива или из другого, ранее созданного объекта `String`. Обратите внимание, что C++ не позволяет делать это с массивами: массивы C++ реализуют семантику ссылок, а не семантику значений.

Можно добавить в класс столько арифметических операций, сколько требуется (например, для сложения объектов `String`, их вычитания, умножения и т. д.). Между тем не нужно забывать о приложении программиста.

Листинг 11.7. Класс `String` с перегруженной операцией присваивания

```
#include <iostream>
using namespace std;

class String {
  char *str;                                  // динамически распределяемый символьный массив
  int len;
  char* allocate(const char* s)               // закрытая функция
  { char *p = new char[len+1];                 // выделение динамической памяти для объекта
    if (p==NULL) exit(1);                     // проверка на успех; выход в случае неудачи
    strcpy(p,s);                              // копирование текста в динамическую память
      return p; }                             // возврат указателя на динамическую память
public:
  String (int length=0);                      // конструктор преобразования/по умолчанию
  String(const char*);                        // конструктор преобразования
  String(const String& s);                    // конструктор копирования
  ~String ();                                 // освобождение динамической памяти
  void operator += (const String&);           // конкатенация с другим объектом
  String operator = (const String&);          // операция присваивания
  void modify(const char*);                   // изменение содержимого массива
  bool operator == (const String&) const;                  // сравнение содержимого
  const char* show() const;                   // возврат указателя массива
  } ;

String::String(int length)
{ len = length;
  str = allocate(""); }                       // копирование пустой строки в динамическую память

String::String(const char* s)                 // конструктор копирования
{ len = strlen(s);                            // определение длины исходного текста
  str = allocate(s);                          // выделение памяти, копирование текста
  cout << " Созданный: '" << str << "'\n"; }

String::String(const String& s)               // конструктор копирования
{ len = s.len;                                // определение длины исходного текста
  str = allocate(s.str);                      // выделение памяти, копирование текста
  cout << " Скопированный: '" << str << "'\n"; }

String::~String()
{ delete str; }                               // возврат динамической памяти (не указателя)

void String::operator += (const String& s)                 // параметр-ссылка
{ len = strlen(str) + strlen(s.str);          // защита от переполнения
  char* p = new char[len + 1];                // выделение достаточного объема памяти
  if (p==NULL) exit(1);                       // проверка на успех
  strcpy(p,str);                              // копирование первой части результата
  strcat (p,s.str);                           // добавление второй части результата
  delete str;                                 // важный шаг
  str = p; }                                  // теперь p может исчезнуть

String String::operator = (const String& s)
{ if (s == this) return *this;                // проверка на самоприсваивание
  delete str;                                 // это нельзя сделать в конструкторе копирования
  len = s.len;                                // копирование данных, отличных от указателя
  str = allocate(s.str);                      // выделение памяти, копирование входящего текста
  cout << " Присвоенный: '" << str << "'\n";          // только для отладки
  return *this; }                             // возврат результата клиенту
```

```
bool String::operator==(const String& s) const          // сравнение содержимого
{ return strcmp(str,s.str)==0: }                        // при совпадении strcmp возвращает 0

const char* String::show() const                        // защита данных от изменений
{ return str; }

void String::modify(const char a[])                     // передача по значению
{  strcpy(str,a,len-1);                                 // защита от переполнения
   str[len-1] = 0; }

String enterData()
{  cout << "Введите название города для поиска: ";     // запрос пользователю
   char data[200];                                      // грубое решение
   cin >> data;                                         // принять ввод от пользователя
   return String(data); }                               // вызов конструктора

int main()
{  cout <<endl << endl;
   enum { MAX = 4 } ;
   String data[4];                                      // база данных объекта
   char *c[4] = { "Атланта", "Бостон", "Чикаго", "Денвер" };
   for (int j=0; j<MAX; j++)
   { data[j] = c[j]; }                                  // присваивание:
data[j].operator+=(c[j]);
   String u; int i;
   u = enterData();                                     // нужно присваивание, нет
                                                        // конструктора копирования
   for (i=0; i < MAX; i++)                              // i определено вне цикла
   { if (data[i] == u) break; }                         // выход, если строка найдена
(data[i].operator==(u))
   if (i == MAX)
     cout << "Город << u.show() << " не найден\n";
   else
     cout << " Город << u.show() << " найден\n";
   return 0;
   }
```

Вопросы производительности

Если нужно инициализировать один объект с помощью другого объекта (при определении, передаче параметров по значению или возврате значения из функции), то следует предусмотреть конструктор копирования. Если требуется присвоить один объект другому, используйте перегруженную операцию присваивания.

Проблемы целостности программы, которые могут возникать из-за динамического управления памятью, настолько опасны, что многие программисты реализуют конструктор копирования и операцию присваивания для каждого класса, работающего с динамической памятью. Часто они это делают даже для класса, не управляющего памятью динамически.

Предотвратить появление проблем можно следующим образом. Разработчики должны изучить требования клиентской программы и понять последствия различных решений.

Существует также ряд проблем, связанных с реализацией в классе избыточного числа функций-членов. Одна из них — слишком объемный исходный код класса. Когда программист, сопровождающий программу (или занимающийся клиентской частью), просматривает все эти бесполезные функции, он отвлекается от более важных деталей. Еще одна проблема — производительность. Как видно

на рис. 11.18, могут возникнуть проблемы с производительностью. Каждому присваиванию вводимой строки в цикле соответствуют два вызова функций и операция присваивания:

1. Вызов конструктора преобразования для параметра функции `operator=()`
2. Вызов самой функции `operator=()`
3. Вызов конструктора копирования для возврата по значению из операции присваивания

Несмотря на все усилия, сохраняется большая разница между объектами класса и встроенными значениям. В данном цикле, если массивы `data[]` и `c[]` имеют компоненты встроенных типов, оператор будет только один. Для класса `String` все иначе: тело цикла представляет три вызова функции:

```
for (int j=0; j<MAX; j++)
{ data[j]=c[j]; }        // присваивание: data[j].operator=(String(c[[j]));
```

Обратите внимание, что все эти операции обходятся достаточно дорого. Кроме самого вызова функции, каждая из них влечет за собой копирование параметра-строки в динамически распределяемую область памяти, вызов конструктора, возврат системе динамически распределяемой памяти. Однократное выполнение таких операций неизбежно для присваивания, поддерживающего для раздельных областей динамической памяти операндов семантику значений. Многократное их повторение для параметра операции присваивания и ее возвращаемого значения кажется чрезмерным. К тому же объект, генерируемый конструктором копирования, не используется клиентом и уничтожается после вызова деструктора.

Первое решение: больше перегруженных операций

Есть два способа, позволяющих повысить производительность перегруженной операции присваивания. Изменение типа параметра операции присваивания со `String` на символьный массив дает возможность исключить вызов конструктора преобразования.

```
String String::operator=(const char s[])              // массив как параметр
{ delete str;              // этого не следует делать в конструкторе копирования
  len = strlen(s);
  str = allocate(s);       // выделение памяти, копирование входящего текста
  cout << "Присвоен: '" << str << " '\n";              // для отладки
  return *this; }
```

```
Присвоен: 'Атланта'
Скопирован: 'Атланта'
Присвоен: 'Бостон'
Скопирован: 'Бостон'
Присвоен: 'Чикаго'
Скопирован: 'Чикаго'
Присвоен: 'Денвер'
Скопирован: 'Денвер'
Введите название города для поиска: Атланта
Создан: ' Атланта '
Присвоен: ' Атланта '
Скопирован: ' Атланта '
Город Атланта найден
```

Рис. 11.19. *Результат программы из листинга 11.7*

Если нужно поддерживать операцию присваивания для символьных массивов и для объектов `String`, следует дважды перегрузить операцию присваивания: для объекта `String` и для символьного массива, который выступает в роли параметра. Результат программы из листинга 11.7 со второй операцией присваивания показан на рис. 11.19. В отладочном сообщении второй операции присваивания добавлено несколько пробелов. Вы можете различать сообщения, выведенные первой операцией присваивания (с параметром типа `String`), и второй операцией присваивания (с параметром типа символьного массива).

Второе решение: возврат по ссылке

Второй способ повышения производительности состоит в исключении лишних вызовов конструктора копирования. Для этого следует заменить возврат по значению на возврат по ссылке. Ниже приведена операция присваивания с параметром типа символьного массива.

```
String& String::operator = (const char s[])                    // возврат ссылки
{ delete str;              // этого не следует делать в конструкторе копирования
  len = strlen(s);
  str = allocate(s);       // выделение памяти, копирование входящего текста
  cout << "Присвоен: '" << str << " '\n";                       // для отладки
  return *this; }
```

```
Присвоен: 'Атланта'
Присвоен: 'Бостон'
Присвоен: 'Чикаго'
Присвоен: 'Денвер'
Скопирован: 'Денвер'
Введите название города для поиска: Денвер
Создан: 'Денвер'
Присвоен: 'Денвер'
Город Денвер найден
```

Рис. 11.20. *Результат программы из листинга 11.7 с добавленной второй операцией присваивания, возвращающей ссылку на объект* `String`

То же самое следует реализовать для первой операции присваивания с параметром `String`. При возврате ссылок из функций убедитесь, что ссылка все еще указывает на объект в левой части операции присваивания в области действия клиента, как например `data[i]` в приведенном выше цикле. Он располагается после операции присваивания, так как определен в области действия клиента. Нужно аккуратно возвращать ссылки на объекты, определенные в области действия сервера — после вызова они исчезают. В этом случае лишь некоторые компиляторы предупреждают вас о возможных последствиях принятия такого решения.

Результат программы из листинга 11.7 с двумя операциями присваивания и возвращаемыми ссылками на объекты см. на рис. 11.20.

Некоторые "блюстители нравов" могут настаивать на том, что было сделано недостаточно, поскольку данная конструкция не запрещает программисту, разрабатывающему код клиента, в частности изменять содержимое возвращаемого объекта-строки перед его уничтожением. Например, в C++ для операций присваивания из листинга 11.7 допустим следующий фрагмент кода.

```
for (int j=0; j<MAX; j++)
{ (data[j] = c[j].modify("Город, о котором никто не слышал"); }
                                                              // допустимо
```

Один объект присваивается другому, возвращается ссылка на целевой объект и немедленно передается сообщение для его модификации. Присвоенное значение никогда не используется. Подобные действия нужно пометить как синтаксическую ошибку. Чтобы сгенерировать синтаксическую ошибку, сделайте возвращаемую ссылку ссылкой на константу.

```
constString& String::operator = (const char s[])          // слишком много?
{ delete str;            // этого не следует делать в конструкторе копирования
  len = strlen(s);
  str = allocate(s);     // выделение памяти, копирование входящего текста
  cout << "Присвоен: '" << str << " '\n";                   // для отладки
  return *this; }
```

Трудно настаивать на именно таком решении, но некоторые сторонники "чистоты нравов" предпочли бы данный вариант.

Практические вопросы: что подлежит реализации

Рекомендуем вам внимательно и аккуратно работать с динамическим управлением памятью. Помните, что вы можете повлиять на производительность программы или ее целостность.

Многие программисты считают, что при разработке класса с динамическим управлением памятью нужно снабдить этот класс полным набором вспомогательных функций-членов:

- Конструктором по умолчанию
- Конструкторами преобразования
- Конструктором копирования
- Перегруженными операциями присваивания
- Деструктором

Не уверен, что необходимо автоматически следовать этой рекомендации. В зависимости от требований клиента может понадобиться лишь часть данных функций. Если вы предусмотрите операции с неподходящими интерфейсами, то проблемы целостности не возникнет, но без всяких на то оснований ухудшится производительность программы. Вы должны разбираться в вопросах, о которых рассказывалось в данной главе. Тогда вы сможете выбирать функции-члены согласно требованиям клиента и создавать корректные и эффективные классы. Если же автоматически следовать приведенным выше рекомендациям, то код клиента будет прекрасно работать, но вы забудете о различиях между инициализацией и присваиванием. А это опасно.

Убедитесь, что вы используете для своих классов подходящие инструменты. При возникновении трудностей проанализируйте ситуацию, используйте операторы отладки, рисуйте диаграммы, но не перегружайте класс ненужными компонентами. Подбирайте инструментальные средства, соответствующие задаче, и обходите "подводные камни" (конструкторы, операции присваивания и т. д.). Запомните, что конструктор копирования и операция присваивания не могут быть взаимозаменяемыми.

Часто клиенту не нужно инициализировать один объект с помощью другого или присваивать один объект другому. Предположим, что реализуемый класс представляет диалоговое окно. Рассмотрим только один элемент данных, представляющий выводимый в окне текст. Такой класс Window будет аналогичен классу `String`. Он содержит динамически распределяемый символьный массив, деструктор и операцию конкатенации, воспринимающую символьный массив, который отображается в окне и добавляется к его содержимому.

```
class Window {
  char *str;                    // динамически распределяемый символьный массив
  int len;
public:
Window()
{ len = 0; str = new char; str[0] = 0; }          // пустая строка String
~Window()
{ delete str; }                 // возвращает динамически распределяемую память
void operator += (const char[])         // параметр-массив
{ len = strlen(str) + strlen(s);
  char* p = new char[len + 1];           // выделение достаточного объема
                                         // динамически распределяемой памяти
  if (p==NULL) exit(1);
  strcpy(p,str); strcat(p,s);           // формирование данных из компонентов
  delete str; str = p; }
```

```
const char* show() const
{ return str; } } ;                    // указатель на содержимое
```

Конечно, в приложении меньше объектов класса Window, чем объектов String. Кроме того, при создании класса Window он инициализируется пустым содержимым, а данные добавляются в процессе выполнения программы.

Объекты типа этого класса не следует передавать по значению. Что, если программист передаст в клиенте параметр Window по значению, или просто пропустит операцию &, тем самым ненамеренно передав параметр по значению?

```
void display(const Window window)          // не делайте этого!
{ cout << window.show(); }
```

Инициализировать одно окно с помощью другого или присваивать одно другому не имеет смысла.

```
Window w1; w1 += "Welcome, Dear Customer!";          // разумное применение
Window w2 = w1;                                // бессмысленное применение
w2 =w1;                                        // менее, чем разумное применение
display(w2);                                   // пропущено значение: slow
```

Вторая и третья строки в этом фрагменте кода не имеют смысла. Большинству программистов этого не потребуется. Кроме того, функция display() передает свой параметр по значению. Большая часть программистов ничего подобного не напишут, но это не значит, что никто не создаст класс Window без конструктора копирования или операции присваивания. Если программист захочет написать такой фрагмент, возникнет проблема целостности и производительности. Однако сам фрагмент будет вполне допустимым исходным кодом C++.

Следует ли писать для класса Window длинные комментарии, типа: "Уважаемый программист клиентской части, пожалуйста, не инициализируйте объекты Window с помощью других объектов Window. Не присваивайте один объект Window другому объекту Window. И не передавайте объект Window функции по значению, а также не возвращайте его из функции. Из-за этого в программе возможны проблемы". Хорошая подача, но не мешало бы сделать что-нибудь для защиты кода клиента.

Один из способов состоит в том, чтобы добавить к классу конструктор копирования и операцию присваивания. Если программист, занимающийся клиентом, напишет неверный код, то это, по крайней мере, не приведет к проблемам целостности.

Кроме того, можно такой программный код сделать синтаксически некорректным. Очень интересная идея. Для этого класс проектируется таким образом, чтобы подобное использование его объектов клиентом приводило к синтаксической ошибке. Разработчик класса решает, какое именно применение объектов будет некорректным. Тогда даже не потребуются комментарии к классу.

Такой метод не прост. Можно исключить определяемый программистом конструктор копирования и перегруженную операцию присваивания, но система подставит для класса свой собственный конструктор копирования, и предусмотренные системой функции-члены способны привести для класса с динамическим управлением памяти к проблемам целостности. Чтобы предотвратить это, добавьте к классу определяемый программистом конструктор копирования и перегруженную операцию присваивания. Рекомендуем вам сделать так, чтобы клиент не мог их использовать и попытка вызова давала синтаксическую ошибку.

Советуем вам написать функцию, которую не может вызывать клиент. Сделайте ее закрытой (или защищенной) функцией.

Это решение показано в листинге 11.8. Конструктор копирования и операция присваивания определены здесь как закрытые. Их даже не нужно реализовывать.

Если задается только прототип функции и функция вызывается клиентом, компоновщик даст ошибку. Он не видит код. Компилятор сообщит, что последние три строки функции main() ошибочны. Закомментируйте объявления для операции присваивания и конструктора копирования, и компилятор безропотно примет клиентский код.

Листинг 11.8. Пример использования закрытых прототипов, чтобы некорректное использование объектов было незаконным

```
#include <iostream>
using namespace std;

class Window {
   char *str;                              // динамически распределяемый символьный массив
   int len;
Window(const Window& w);                   // закрытый конструктор копирования
Window& operator = (const Window &w);      // закрытое присваивание
public:
Window()
{ len = 0; str = new char; str[0] = 0; }   // пустая строка String
~Window()
{ delete str; }                            // возвращает динамически распределяемую память
void operator += (const char s[])          // параметр-массив
{  len = strlen(str) + strlen(s);
   char* p = new char[len + 1];             // выделение достаточного объема
                                           // динамически распределяемой памяти
   if (p==NULL) exit(1);
   strcpy(p,str); strcat(p,s);             // формирование данных из компонентов
   delete str; str = p; }
const char* show() const
{ return str; } } ;                        // указатель на содержимое

void display(const Window window)          // не передавать объекты по значению
{ cout << window.show(); }

int main()
{  Window w1; w1 += "Добро пожаловать, уважаемые покупатели!\n";          // разумно
   Window w2 = w1;                         // неразумное использование: синтаксическая ошибка
   w2 = w1;                                // еще менее разумно: синтаксическая ошибка
error
   display(w2);                            // передача по значению: синтаксическая ошибка
   return 0;
   }
```

Этот способ защитит ваши классы от постоянного использования их программистами клиентской части. Если клиентский код, помечаемый в функции main() листинга 11.8 как необдуманный, по какой-то причине нужно поддерживать и важна производительность программы, класс должен предусматривать конструктор копирования и операцию присваивания или несколько операций присваивания. Следует предусмотреть операции преобразования, если объекты класса должны инициализироваться из простых объектов данных, а не из объектов того же типа. Еще одна веская причина для добавления операций преобразования состоит в том, что они позволяют избежать определения нескольких перегруженных операторных функций. За счет этого уменьшается число функций в классе, но потребуются дополнительные вызовы конструкторов и добавочные операции по распределению памяти.

Итоги

В этой главе рассматривалась "темная сторона" мощных возможностей C++. Сделано это не для того, чтобы пугать читателя, а чтобы внушить ему чувство ответственности за производительность и целостность программы C++.

Здесь пришлось еще раз покритиковать передачу параметров по значению. Передавайте параметры по ссылке и используйте модификатор `const`, когда параметр в функции не изменяется.

Кроме того, приводились аргументы против возврата объектов из функций по значению. Если нужно вернуть из функции объект, возвращайте ссылку на него, но убедитесь в том, что это ссылка на объект, который не исчезнет сразу после вызова функции.

Если вы убеждены, что код клиента не должен передавать объекты вашего класса по значению, сделайте конструктор закрытым, т. е. поместите прототип конструктора в закрытую секцию класса. Советуем вам не реализовывать саму функцию.

Если класс управляет памятью динамически, снабдите его деструктором, возвращающим динамически распределяемую память.

Если объекты класса должны использоваться в клиенте для инициализации других объектов класса, применяйте конструктор копирования, реализующий для класса семантику значений и снабжающий каждый объект собственной областью динамически распределяемой памяти.

Если объекты класса требуется присваивать в клиенте друг другу, предусмотрите перегруженную операцию присваивания, реализующую семантику значений и предоставляющую каждому объекту собственную область динамической памяти. В операции присваивания не забудьте о предотвращении "утечек памяти": нужно возвращать используемую объектом ранее память перед присваиванием нового значение, переданного в параметре-объекте. Определитесь с тем, нужно ли поддерживать цепочку присваиваний. Часто в клиенте она не требуется.

Применение конструкторов преобразования позволяет значительно ослабить правила строгого контроля за типами в C++. В качестве фактического аргумента можно передавать данные, тип которых отличается от типа, требуемого классом. При этом программный код все равно будет вполне законным. Хороший метод, но пользоваться им нужно аккуратно. Дополнительные вызовы конструкторов преобразования — это накладные расходы, особенно если приходится поддерживать семантику значений.

Нужно различать, где в клиенте могут вызываться конструкторы копирования, а где операция присваивания. Для обозначения операции в обоих случаях используется знак равенства, но вызываются разные серверные функции. Необходимо знать, какая именно функция вызывается.

Возвращайтесь к материалу данной главы. Рисуйте диаграммы использования памяти, экспериментируйте с примерами программ. К сожалению, в C++ к традиционным категориям ошибок — синтаксическим и семантическим (этапа выполнения) — добавляется еще одна: программа может быть синтаксически и семантически корректной, но при этом все равно неверной. Относитесь к этому языку с должным уважением. Удачи вам.

Часть III

Объектно-ориентированное программирование с агрегированием и наследованием

В этой части книги обсуждаются методы объектно-ориентированного программирования. Рассматриваются инструментальные средства программиста: композиция и наследование классов. Некоторые программисты не знают, какой метод выбрать и как избежать чрезмерного усложнения программы.

В главе 12 описывается синтаксис использования объектов как компонентов другого класса, определяются правила доступа к таким объектам и их элементам данных, поясняется, как инициализировать компоненты составного объекта. Кроме того, эта глава знакомит читателей с методами совместного использования компонентов объектов как статических компонентов и с помощью компонентов-ссылок, описывается применение вложенных и "дружественных" классов.

В главе 13 представлены методы использования наследования. Рассказывается о синтаксисе наследования C++, обсуждаются различные режимы наследования и их влияние на получаемый в результате объект. Кроме того, в ней определяются права доступа к базовым компонентам производного объекта, правила области действия с учетом наследования и поясняются правила разрешения имен, когда производный метод скрывает базовый метод с тем же именем. Рассматриваются также правила создания и уничтожения производных объектов, описывается последовательность вызова конструкторов и деструкторов.

В главе 14 рассматривается унифицированный язык моделирования — UML (Unified Modeling Language). Он становится все более популярным в объектно-ориентированном программировании и применяется для описания проектов. Эта глава поможет читателю сделать выбор между наследованием и композицией класса и подобрать подходящие инструменты для разработки классов. Кроме того, здесь рассказывается об опасностях чрезмерного применения наследования и значительного усложнения программ.

Глава 12

Преимущества и недостатки составных классов

Темы данной главы

- ✔ Использование объектов классов как элементов данных
- ✔ Инициализация составных объектов
- ✔ Элементы данных со специальными свойствами
- ✔ Контейнерные классы
- ✔ Итоги

В первых двух частях этой книги говорилось в основном о правилах языка C++. Вы узнали, что может делать программист, чего нужно избегать, чтобы программа не перестала функционировать. C++ был представлен как мощный язык, при работе с которым от программиста ожидается глубокое понимание того, что происходит внутри программы и лежит "на поверхности".

Во второй части книги определились базовые принципы объектно-ориентированного программирования, связанного с написанием программ на C++, анализом взаимодействия между классами программы. Описывались следующие идеи:

- Связывание данных и функций в классе для того, чтобы показать их логическое единство.
- Определение как закрытых тех компонентов класса, которые делают клиентов зависимыми от низкоуровневых деталей архитектуры класса.
- Использование области действия класса как дополнительного инструмента, устраняющего конфликт между элементами разных классов и необходимость согласования действий между разработчиками классов.
- Создание функций-членов, делающих излишним прямой доступ к именам полей данных сервера в клиенте.
- Перенос обязанностей с клиента на серверные классы и функции-члены.
- Написание клиентской части в терминах вызова методов сервера, в результате чего в программе не создаются зависимости от архитектуры сервера.

- Создание конструкторов и деструкторов для правильной инициализации объектов, управления ресурсами и для дальнейшего переноса обязанностей на серверные классы.
- Передача программисту, сопровождающему приложение, и программистам, отвечающим за клиентскую часть, идей разработчика и его знания поведения сервера, например, с помощью модификаторов `const`, применяемых к элементам данных, параметрам, возвращаемым значениям и методам.

Эти идеи лежат в основе базовой техники программирования, которая выражается в "самодокументируемом" объектно-ориентированном коде. Такой исходный код прост в понимании и его легко сопровождать. С помощью данных идей можно полностью реализовать потенциал C++. Без них программа будет состоять из сильно связанных друг с другом фрагментов с большим числом зависимостей. Такой программный код труден в понимании и модификации, причем независимо от того, на каком языке он написан — на C++, Java, COBOL или FORTRAN.

В этой части книги рассматривается проектирование программ, содержащих несколько взаимодействующих классов, изучается *композиция классов*, когда объекты одного класса используются как элементы данных, локальные переменные или параметры другого класса. Это мощная техника организации взаимодействия между классами программы. Архитектурные решения, которые нужно реализовать с помощью композиции классов, поддерживаются правилами вызова конструкторов C++ и передачи данных из клиента в компоненты определяемых программистом классов.

Еще одним методом кооперации между классами является наследование, позволяющее проектировать похожие классы,— один класс дополняет элементы данных и методы другого класса. Это основной способ повторного использования программного кода в C++. Здесь обсуждаются вопросы проектирования и применения наследования в той или иной ситуации, рассматривается множество средств языка C++, которые используются для поддержки наследования: синтаксис наследования, экземпляры объектов, передача данных для инициализации наследуемых компонентов, неоднозначность имен и правила разрешения этой неоднозначности.

Программисты, работающие с C++, любят применять наследование. Многие эксперты считают, что использование наследования является основой объектно-ориентированного программирования. Это не совсем так. Основа объектно-ориентированного программирования — использование классов как фундамента объектно-ориентированной программы (для связывания данных и операций, управления доступом к компонентам и т. д.).

Наследование не является фундаментом объектно-ориентированного программирования. Это техника написания программного кода и его повторного использования. В таком качестве она очень важна в C++, поэтому применять ее следует корректно.

Использование объектов классов как элементов данных

Основная цель конструктора класса в C++ — позволить программисту связать вместе логически соотнесенные данные и операции (см. главу 9).

Почти все примеры классов C++, встречавшиеся ранее в этой книге, включали в себя элементы данных встроенных типов — целые и с плавающей точкой. В некоторых более сложных примерах использовались массивы символов. Фактически это были указатели на массивы символов, распределяемые в динамической области памяти. С точки зрения композиции классов, указатель аналогичен целым

значениям и числам с плавающей точкой. Он не имеет доступной извне внутренней структуры. Между тем компоненты классов могут быть более сложными, чем значения встроенных типов.

В главах 10 и 11 было показано, что в языке C++ большое внимание уделяется равноценной интерпретации встроенных типов и типов, определяемых программистом. Если данные встроенных типов можно использовать как компоненты классов, то нет никаких причин для запрещения применять в качестве элементов данных объекты некоторых других классов, также содержащие компоненты.

C++ позволяет использовать объекты классов как компоненты объектов других классов. Если один класс содержит множество элементов данных, можно объединить группу родственных данных в один объект, объявив его компонентом класса. Вместо небольшого числа крупных классов со многими компонентами получится большое число классов с меньшим количеством компонентов. Каждый программист в процессе разработки будет заниматься своим делом. Кроме того, применение большого числа более мелких классов повышает модульность программы, содействует сокрытию ненужных деталей от клиента. Недостатком такой чрезмерной модульности является то, что клиентам придется иметь дело с большим числом мелких классов, а это затрудняет их изучение.

Классы, в состав которых в качестве элементов данных входят объекты других классов, называются *составными* или *композитными классами*. Почти все классы содержат компоненты (элементы данных), следовательно, являются составными, однако этот термин применяется в основном к классам, компоненты которых содержат, в свою очередь, собственные компоненты. В теории объектно-ориентированного проектирования использование объектов одного класса в качестве компонентов другого класса называется *агрегированием* или *композицией класса*.

Например, рассмотрим класс Rectangle, содержащий координаты x и y соответственно верхнего левого и нижнего правого углов прямоугольника (общепринятое соглашение в программировании графических приложений).

```
class Rectangle {
  int x1, y1;                                  // координаты верхней левой точки
  int x2, y2;                                  // координаты нижней правой точки
  int thickness;                               // толщина границы прямоугольника
public:
  Rectangle (int inX1, int inY1, int inX2, int inY2, int width=1);
  void move(int a, int b);                     // перемещение прямоугольника
  void setThickness(int width = 1);            // изменение толщины
  bool pointIn(int x, int y) const;            // точка в прямоугольнике?
  . . . . } ;                                  // остальная часть Rectangle

Rectangle::Rectangle (int inX1, int inY1, int inX2, int inY2, int width)
{ x1 = inX1; y1 = inY1;
  x2 = inX2; y2 = inY2;
  thickness = width; }                         // установка элементов данных

void Rectangle::move(int a, int b)
{ x1 += a; y1 += b;
  x2 += a; y2 += b; }                          // перемещение каждого угла

void Rectangle::setThickness(int width)
{ thickness = width; }                        // выполнение работы

bool Rectangle::pointIn(int x, int y) const          // точка внутри?
{ bool xIsBetweenBorders = (x1<x && x<x2 || (x2<x && x<x1);
  bool yIsBetweenBorders = (y>y1 && y<y2) || (y<y1 && y>y2);
  return (xIsBetweenBorders && yIsBetweenBorders); }
```

Данный класс предоставляет клиентам возможность перемещения объекта-прямоугольника по экрану, изменения толщины линии границы прямоугольника и проверки попадания точки внутрь прямоугольника. Кроме того, он может определить объекты класса Rectangle путем спецификации координат углов. Он перемещает точку и прямоугольник по экрану, пытаясь "поймать" ее в прямоугольник.

```
int x1=20,y1=40; int x2=70,y2=90;          // левый верхний/правый нижний углы
int x=100, y=120;                          // точка для отлова в прямоугольник
Rectangle rec(x1,y1,x2,y2,4);              // создание объекта Rectangle
rec.setThickness();                  // толщина линии - 1 пиксель (по умолчанию)
x -= 25; y -= 15;                          // перемещение точки по экрану
rec.move(10,20);                           // 10 пикселей вправо, 20 вниз
if (re.pointIn(x,y)) cout << "Точка внутри\n";     // точка в прямоугольнике?
```

Внутренняя структура класса Rectangle является сложной структурой. При написании этого фрагмента намеренно сделаны ошибки: перепутаны x1 и y1, x2 и y2 и т. д. Программист, занимающийся клиентской частью, сразу поймет, что для спецификации объекта Rectangle (пять значений в вызове конструктора) требуется много работы. Причина такой сложности класса Rectangle и его клиента состоит в отсутствии реализации компонента — класса Point (точка). Концепция точки здесь выглядит вполне естественно, она даже присутствует в комментариях класса Rectangle и его клиента, но не поддерживается с помощью определяемого программистом типа.

Синтаксис C++ для композиции класса

Рассмотрим тот же пример, однако обратим внимание на класс Point, предлагающий клиенту некоторые новые сервисы. Представим, что это фрагмент большой программы, над разными частями которой работает множество людей. Сосредоточьтесь на синтаксисе композиции класса и на вопросах, связанных со взаимодействием между классами и их разработчиками.

```
class Point {
private:
  int x, y;                                  // закрытые координаты
public:
  Point (int a, int b)                       // обобщенный конструктор
    { x = a; y = b; }
  void set (int a, int b)                    // функция-модификатор
    { x = a; y = b; }
  void move (int a, int b);                  // функция-модификатор
    { x += a; y += b; }
  void get (int& a, int& b) const            // функция-селектор
    { a = x; b = y; }
  bool isOrigin () const                     // функция-предикат
    { return x == 0 && y == 0; } } ;
```

Для функций-членов здесь используется общепринятая терминология. *Модификатор* — это функция-член, изменяющая состояние целевого объекта (вы заметили отсутствие ключевого слова const?). *Селектор* — функция-член, не изменяющая состояние целевого объекта (как видно, ключевое слово const в ней присутствует). *Предикат* представляет собой селектор, возвращающий булево значение, которое сообщает о состоянии целевого объекта (в данном случае говорит о том, что точка является началом координат).

В этом примере применяются распространенные имена функций-членов. Тем самым иллюстрируется, что границы класса эффективно ограничивают конфликты имен в программе. При выборе имени set() для функции-члена класса Point об

этом не нужно уведомлять всех разработчиков других классов приложения. Они также могут использовать в своих классах имя set(). Об имени set() должны знать лишь те, кто использует класс Point в качестве сервера. Одним из таких клиентских классов является класс Rectangle, представленный в начале главы. В показанной ниже версии класса Rectangle имеются два элемента данных класса Point, обозначающих верхний левый и нижний правый углы прямоугольника. Элемент данных thickness обозначает толщину линии при выводе прямоугольника на экран.

```
class Rectangle {
  Point pt1, pt2;                          // координаты верхней левой точки
  int thickness;                           // толщина границы прямоугольника
public:
  Rectangle (int inX1, int inY1, int inX2, int inY2, int wid=1);
  void move(int a, int b);                 // перемещение прямоугольника
  void setThickness(int width = 1);        // изменение толщины
  bool pointIn(int x, int y) const;        // точка в прямоугольнике?
  . . . . . } ;                            // остальная часть Rectangle

Rectangle::Rectangle (int inX1, int inY1, int inX2, int inY2, int width)
{ pt1.set(inX1,inY1); pt2.set(inX2,inY2);   // перенос обязанностей на сервер
  thickness = width; }                      // установка элементов данных

void Rectangle::move(int a, int b)
{ pt1.move(a,b); pt2.move(a,b); }           // перемещение каждого угла

void Rectangle::setThickness(int width)
{ thickness = width; }                      // выполнение работы

bool Rectangle::pointIn(int x, int y) const         // точка внутри?
{ int x1,y1,x2,y2;                         // координаты углов прямоугольника
  pt1.get(x1,y1); pt2.get(x2,y2);          // получение данных точки
  bool xIsBetweenBorders = (x1<x && x<x2 || (x2<x && x<x1);
  bool yIsBetweenBorders = (y>y1 && y<y2) || (y<y1 && y>y2);
  return (xIsBetweenBorders && yIsBetweenBorders); }
```

Здесь класс Rectangle изменился. Эти изменения представляют общие идиомы программирования для композиции класса.

Вместо набора присваиваний нижнего уровня многочисленным элементам данных встроенных типов в конструкторе Rectangle передаются всего лишь два сообщения объектам-компонентам.

```
pt1.set(inX1,inY1), pt2.set(inX2,inY2);    // перенос обязанностей на сервер
```

Это пример изоляции клиента (класса Rectangle) от деталей архитектуры сервера (класса Point). Клиент написан в терминах передачи сообщений серверным объектам. Он не использует детали архитектуры сервера нижнего уровня. В коде клиента говорится, *что* нужно сделать. Детали операций перенесены из класса Rectangle в класс Point. Такой стиль упрощает понимание клиентской программы.

Метод move() представляет еще более интересную идиому C++ для связи между составными и компонентными классами. Когда у объекта Rectangle затребовано перемещение, он просит свои компоненты проделать необходимые действия, вызывая метод с тем же именем move(). Второй метод move() принадлежит к компонентному классу Point, а не к составному классу Rectangle. Таким образом, здесь показан еще один пример равнозначной интерпретации в C++ объектов разной природы. В данном случае методы с одним и тем же именем принадлежат классам с аналогичным поведением. Перемещение прямоугольника означает перемещение каждой его точки. Возможность писать методы, передающие информацию своим элементам данных,— одна из причин, почему оба метода можно называть move(), а не movePoint() и moveRectangle().

Доступ к элементам данных элементов данных класса

Еще одно важное отличие этой архитектуры класса Rectangle от его предыдущей версии состоит в доступе к компонентам компонентов класса. В предыдущей версии класс Rectangle мог делать со своими координатами *x* и *y* все что угодно. Они были непосредственно доступны. В последней версии Rectangle они являются компонентами класса Point. Если бы компонентный объект (в данном примере класса Point) имел общедоступные компоненты, то составной класс (Rectangle) мог бы обращаться к элементам данных своего компонентного объекта с помощью операции-селектора (точки). Если бы компоненты класса Point были общедоступными, функция-член Rectangle::pontIn() могла бы использовать уточненные имена компонентов Point. Таким образом, класс Point может определить, находится ли параметр x между *x*-координатами элементов данных Rectangle — pt1 и pt2.

```
bool xIsBetweenBorders = (pt1.x<x && x <pt2.x)
            || (pt2.x<x && x<pt1.x)
```

Элементы данных Point являются закрытыми, а у клиентского класса (в примере это Rectangle) нет специальных полномочий на доступ к компонентам сервера (Point). Класс Rectangle использует объекты Point как свои собственные компоненты. Следовательно, его методы могут обращаться к элементам данных Point (pt1 и pt2). Однако методы класса Rectangle не могут обращаться к элементам данных x и y класса Point. Написанная выше строка не является допустимой. Для обращения к компонентам Point в методах Rectangle используйте общедоступные функции-члены, например Point::get().

Не путайте два разных контекста. Класс Rectangle может без ограничений обращаться к своим собственным закрытым компонентам Point pt1 и pt2, но не к закрытым компонентам своих элементов данных pt1.x, pt1.y, pt2.x и pt2.y. Именно поэтому Rectangle::pointIn() использует данный код для получения элементов данных Rectangle pt1 и pt2.

```
bool Rectangle::pointIn(int x, int y) const
{ int x1,y1,x2,y2;                              // координаты углов
  pt1.get(x1,y1); pt2.get(x2,y2);               // получить данные точки
  bool xIsBetweenBorders = (x1<x && x<x2) || (x2<x && x<x1);
  . . . }                                       // и т. д.
```

Необходимость использовать серверные функции для доступа к элементам данных класса часто раздражает. Это может сделать работу по проектированию методов составных классов весьма трудоемкой и утомительной.



Осторожно! Если класс содержит элементы данных, принадлежащие другим классам, то функции-члены класса не могут обращаться к закрытым компонентам этих элементов данных. Составной класс должен использовать для доступа к таким элементам данных методы, которые к ним обращаются.

Рассмотрев архитектуру клиента класса Rectangle, можно заметить, что для нее не нужны никакие изменения. Клиент должен подставлять пять аргументов конструктору Rectangle и два аргумента методу Rectangle::pointIn(). Это означает, что введение компонента класса Point положительно отражается на архитектуре составного класса Rectangle, но не дает преимуществ в плане архитектуры клиента.

```
int x1=20, y1=40; int x2=70, y2=90;    // углы прямоугольника
int x=100, y=120;                      // точка, отлавливаемая в прямоугольнике
```

```
Rectangle rec(x1,y1,x2,y2,4);       // создание объекта Rectangle
rec.setThickness();                 // линия толщиной в пиксель (по умолчанию)
x -= 25; y -= 15;                   // перемещение точки по экрану
rec.move(10,20);                    // 10 пикселей вправо, 20 вниз
if (rec.pointIn(x,y)) cout << "Точка внутри\n";    // точка в прямоугольнике?
```

В этом маленьком примере различия также невелики и понятны. Подобно первой версии класса Rectangle, этот клиент выполняет обработку в терминах отдельных сущностей x и y. Данный код не агрегирует их в класс и не передает сопровождающему приложение программисту замыслы проектировщика, не сообщает, как соотносятся отдельные элементы, и ничего не говорит о том, что они представляют координаты одной точки. Если клиенту потребуются две точки, как при перемещении, сравнении и т. д., то эти действия следует реализовывать в клиенте через отдельные координаты. Такие индивидуальные операции низкого уровня усложняют клиентскую часть и затрудняют понимание смысла действий. Возьмем оператор клиента `rec.move(10,20);`. Здесь ясно видно, что прямоугольник перемещается. То, что должна перемещаться точка с координатами (100,120), придется уяснить из серии присваиваний `x -= 25; y -= 15;`. Эти обязанности низкого уровня не перенесены на уровень сервера.

Выражение клиента в терминах объектов класса Point и операций с ними делает программный код более объектно-ориентированным.

```
Point p1(20,40), p2(70,90);      // углы прямоугольника
Point point(100,120);            // точка для отлова в прямоугольнике
Rectangle rec(p1,p2,4);          // ниже рассказывается о возможных проблемах
rec.setThickness();              // линия толщиной в пиксель (по умолчанию)
point.move(-25,-15);             // перемещение точки по экрану
rec.move(10,20),                 // 10 пикселей вправо, 20 вниз
if (rec.pointIn(x,y)) cout << "Точка внутри\n";    // точка в прямоугольнике?
```

Здесь два сервера — классы Point и Rectangle. То, что перемещается точка point класса Point, не менее ясно, чем перемещение объекта rec класса Rectangle. Операции низкого уровня перенесены на серверы, а клиент выражается в терминах вызовов функций.

В этом фрагменте необходимы интерфейсы класса Rectangle, доступ к которым не предусматривается. Класс Rectangle предлагает конструктор с пятью параметрами, а клиент подставляет только три. Класс Rectangle ожидает двух аргументов для функции pointIn(), а клиент допускает один. Проблему можно разрешить, изменив вызовы функций в клиенте или интерфейсы функций в сервере Rectangle.

Если бы класс Rectangle был библиотечным и изменить его было бы невозможно, то клиенту пришлось бы обходить ограничения библиотеки. Если же класс Rectangle представляет собой один из вспомогательных классов, разрабатываемых для приложений, его можно изменить. С точки зрения объектно-ориентированной идеологии именно серверный класс (здесь Rectangle) должен соответствовать ожиданиям и требованиям клиентской части. Класс Rectangle следует переписать следующим образом:

```
class Rectangle {
  Point pt1, pt2;                          // координаты верхней левой точки
  int thickness;                           // толщина границы прямоугольника
public:
  Rectangle (const Point& p1, const Point& p2, int width=1);
  void move(int a, int b);                 // перемещение прямоугольника
  void setThickness(int width = 1);        // изменение толщины
  bool pointIn(const Point& pt) const;     // точка в прямоугольнике?
    . . . . } ;                            // остальная часть Rectangle
```

```
Rectangle::Rectangle (const Point& p1, const Point& p2, int width)
{ pt1 = p1; pt2 = p2;
  thickness = width; }                    // установка элементов данных

void Rectangle::move(int a, int b)
{ pt1.move(a,b); pt2.move(a,b); }         // перемещение каждого угла

void Rectangle::setThickness (int width)
{ thickness = width; }                    // выполнить работу

bool Rectangle::pointIn(const Point& pt) const          // точка внутри?
{ int x,y,x1,y1,x2,y2;                    // координаты углов прямоугольника
  pt.get(x,y);                            // получить координаты параметра
  pt1.get(x1,y1); pt2.get(x2,y2);         // получить оба угла
  bool xIsBetweenBorders = (x1<x && x<x2 || (x2<x && x<x1);
  bool yIsBetweenBorders = (y>y1 && y<y2) || (y<y1 && y>y2);
  return (xIsBetweenBorders && yIsBetweenBorders); }
```

Обратите внимание на ключевое слово const (и его отсутствие) в классах Point и Rectangle. Оно отражает изменение (или отсутствие изменений) в целевом объекте и параметрах вызова функции. Так как параметр функции в этом варианте не возвращает указатели или ссылки на объекты, нет никакой необходимости использовать для возвращаемых значений ключевое слово const.

Обобщенный конструктор класса Rectangle может вызываться с двумя или с тремя параметрами. При вызове с двумя параметрами элемент данных thickness устанавливается в значение по умолчанию (единица).

Доступ к элементам данных параметров метода

Обратите внимание, что параметры метода интерпретируются в C++ аналогично элементам данных составного класса. Параметры функции-члена могут иметь любой тип, включая объекты классов. Ограничения не накладываются на режимы параметров для объектов классов. Они могут передаваться по значению, по ссылке, по указателю и при необходимости могут иметь модификатор const.

При доступе к параметрам-объектам в функции-члене соблюдаются те же правила, что и при доступе к другим объектам. Разрешается обращаться только к компонентам public. Сам параметр доступен методу, но его закрытые компоненты — нет. Если клиенту параметра (функции-члену) нужен доступ к закрытым компонентам сервера, используйте функции-члены серверного класса.

Именно поэтому для доступа к компонентам параметра pt.x и pt.y функция-член pointIn() в Rectangle использует функцию доступа get() класса Point.

Если другой объект, к которому производится доступ, представляет объект *того же* класса, то в C++ имеется важное исключение. Оно применяется, когда объект передается как параметр функции-члена класса, к которому принадлежит этот объект. Если клиентский и серверный классы являются классами одного типа, то объект клиента имеет полные права доступа к компонентам объекта-параметра. Предположим, что нужно добавить к классу Point функцию-член isSamePoint(). С ее помощью сравниваются координаты целевого объекта и объекта-параметра, возвращается true, если они содержат одинаковые значения, и false в противном случае.

```
bool Point::isSamePoint(const Point& p) const          // сравнение данных
{ return x==p.x && y == p.y; }
```

По существу, доступ к другому экземпляру объекта (в данном случае — p в isSamePoint()) происходит в области действия целевого объекта (типа Point). Следовательно, он разрешается.



Советуем Когда параметр метода класса имеет тип компонента класса, метод не может непосредственно обращаться к частным компонентам параметра и должен использовать для этого компонентные функции параметра. Если же параметр метода класса имеет тот же тип, что и сам класс, то метод может обращаться к частным компонентам параметра без помощи функций доступа. В этом случае применение функций доступа будет синтаксически корректно, но некрасиво.

Согласно правилу, закрытые компоненты объекта не должны быть доступны извне, поэтому другой компонент того же класса должен использовать функции доступа. Такую функцию можно написать следующим образом:

```
bool Point::isSamePoint(const Point& pt) const          // сравнение данных
{ int x1, y1;
  pt.get(x1,y1);                          // излишне: доступ через функцию-член
  bool answer = (x==x1 && y==y1);
  return answer; }
```

Чтобы избежать такого ненужного, создающего лишние трудности кода, C++ допускает незначительную несогласованность в правах доступа. Данная версия `isSamePoint()` синтаксически корректна. Программист, работа которого оценивается по количеству строк исходного кода, пишет огромное число операторов, и его программа будет выглядеть примерно так. Другой вопрос, как этот объем влияет на качество ПО.

Инициализация составных объектов

Вопросы инициализации играют в программировании важную роль. Отсутствие выполненной должным образом инициализации вычислительного объекта — распространенный источник ошибок. C++ предлагает программистам богатый набор методов инициализации для компонентов программы.

При описании синтаксиса для определения глобальных переменных в главе 3 рассматривалось присваивание переменным начальных значений. В главе 5 при пояснении синтаксиса составных данных (структур, массивов, перечислений, объединений и битовых полей) обсуждалась инициализация этих программных компонентов. Кроме того, говорилось об инициализации объектов.

Все это не случайно. Инициализация объектов C++ — одна из основных забот программиста. Кроме того, это важная часть работы по переносу обязанностей на серверный класс и освобождению клиента от деталей низкого уровня архитектуры сервера. Перейдем к описанию инициализации составных объектов. Аналогично мы поступим при изучении наследования: после обсуждения синтаксиса наследования будет продемонстрирована инициализация производных объектов.

Как уже упоминалось в главе 9, синтаксис C++ является одинаковым для определения переменных и элементов данных класса. Следующая строка интерпретируется как допустимая в области действия функции и в области действия класса.

```
int x,y;                  // может быть в функции или блоке, а может быть и в классе
```

Между тем определение с инициализацией может присутствовать в выполняемом коде только в функции или в блоке.

```
int x=100, y=100;         // может быть в функции или в блоке, но не в классе
```

Следовательно, нельзя инициализировать элементы данных в спецификации класса, используя синтаксис, подходящий для инициализации переменных C++. Это могут делать программисты, работающие с Java, но не с C++.

```
class Point {
  int x=100, y=100;                    // недопустимый синтаксис инициализации
public:
  Point (int a, int b);                // подходящий способ инициализации
    { x = a; y = b; }
    . . . } ;                          // остальная часть класса Point
```

Синтаксис определения экземпляров объектов и элементов данных в составных классах будет таким же. Следующая строка может встречаться как в теле функции, так и в определении класса:

```
Point pt1, pt2;                        // может быть в функции, блоке или классе
```

Подстановка аргументов для инициализации допускается только при определении экземпляров объекта в области действия функции или блока.

```
Point pt1(20,40),pt2(70,90);       // OK в функции или в блоке, но не в классе
```

Следовательно, элементы данных в составных классах не могут инициализироваться в спецификациях класса с помощью синтаксиса, подходящего для инициализации объектов компонентного класса. В следующем примере инициализируются компоненты Point класса Rectangle с применением синтаксиса, подходящего для переменной Point. Компилятор отвергает такой синтаксис.

```
class Rectangle {                      // некорректная спецификация класса
  Point pt1(20,40);                    // допускается в клиенте, но не здесь
  Point pt2(70,90);                    // та же проблема
  int thickness = 1;                   // та же проблема
public:
  Rectangle (const Point& p1, const Point& p2, int width = 1);
  void move(int a, int b);
  . . . . } ;                          // остальная часть класса Rectangle
```

Вместо этого C++ предлагает два способа инициализации компонентов составного класса. Один из них состоит в присваивании значений в теле конструктора компонентного класса. Конструктор может присваивать значения соответствующим элементам данных, будь то компоненты составных типов или компоненты встроенных типов данных.

```
Rectangle::Rectangle (const Point& p1,const Point& p2,int width)
{ pt1 = p1; pt2 = p2;                  // присваивает значения компонентам
                                       // составного типа
  thickness = width; }                 // присваивает значения компонентам
                                       // встроенных типов
```

Другой способ — использовать список инициализации компонентов, вызывающего конструктор компонентов класса. В следующем примере список инициализации компонентов вызывает конструктор Point для инициализации компонентов Point класса Rectangle.

```
Rectangle::Rectangle (const Point& p1, const Point& p2, int width)
  : pt1(p1), pt2(p2);              // вызов конструкторов для компонентов
  { thickness = width; }           // присваивает значения встроенным компонентам
```



Осторожно! Синтаксис определения переменных и объектов в C++ и синтаксис определения компонентов класса одинаков. Однако его нельзя использовать для инициализации переменных и объектов C++ при определении компонентов класса. Следует применять используемый по умолчанию конструктор или список инициализации компонентов.

Сначала обсудим детали инициализации в теле конструктора, а потом перейдем к списку инициализации.

Применение используемых по умолчанию конструкторов компонента

В главе 9 рассказывалось, что при создании объекта C++ для его компонентов выделяется память. Если не углубляться в вопросы выравнивания памяти для хранения значений, можно предположить, что выделенная объекту память равна сумме размеров его элементов данных.

Говорилось также, что при создании объекта C++ его элементы данных инициализируются *в вызове* конструктора, и подчеркивалось, что конструктор вызывается *после* создания всех элементов данных объекта.

Можно допустить, что для классов, которые имеют только несоставные поля встроенных типов, различие между "*во время*" и "*после*" не очень существенно. Оно подобно различию между присваиванием и инициализацией. Для несоставных элементов встроенных типов различие почти незаметно. Как уже показывалось в главе 11, оно может стать очень важным для классов с динамическим распределением памяти. Если не различать их, могут возникнуть проблемы с производительностью и целостностью программы.

Различие между "*во время*" и "*после*" приобретает важность для объектов, компоненты которых являются объектами определяемых программистом классов. В данном разделе рассмотрен процесс создания составного объекта.

При создании объекта C++ выделяется память его элементам данных, а затем выполняется тело конструктора. Это означает, что конструктор для объекта вызывается *после* создания всех элементов данных.

Важной характеристикой этого процесса является то, что элементы данных создаются в порядке их следования в спецификации класса. Когда процесс заканчивается, объект составного типа выглядит в памяти как сумма его компонентов.

Это означает, что изменение порядка компонентов в классе может повлиять на его свойства. Однако имейте в виду, что это возможно лишь в случае, если элементы данных зависят друг от друга. Например, один элемент данных может представлять число компонентов в другом.

Когда элементы данных создаются последовательно, друг за другом, поля данных встроенных типов (если они имеются) либо остаются неинициализированными (когда память для объекта выделяется в стеке или в динамически распределяемой памяти), либо устанавливаются в 0 (для объектов, созданных как глобальные или статические). Если объекту необходимо сохранить в поле встроенного типа конкретное значение, об этом нужно позаботиться при вызове конструктора. Например, поле `thickness` для объектов класса `Rectangle` устанавливается в значение, заданное параметром конструктора `width`.

Что же происходит, если объект представляет собой составной объект, включающий элементы данных, которые являются объектами других классов? Приведенное выше замечание о создании объектов в порядке их следования может сбить с толку. Процедура создания объекта рекурсивна. После создания элемента данных вызывается конструктор.

Вспомните, что ни один объект языка C++ не может создаваться без вызова конструктора, следующего за выделением памяти. Если поле составного объекта содержит данные определяемого программистом типа (структуры или класса), то

конструктор вызывается сразу после выделения памяти для поля и перед созданием следующего поля. После успешного создания всех полей составного объекта выполняется тело конструктора составного класса.

При создании объекта Rectangle события разворачиваются в следующем порядке:

1. Создается элемент данных pt1 класса Point.
2. Создается элемент данных pt2 класса Point.
3. Создается элемент данных thickness типа int.
4. Выполняется тело конструктора класса Rectangle.

Когда в процессе построения объекта Rectangle создается каждый элемент данных класса Point, это происходит так:

1. Создается элемент данных x типа int.
2. Создается элемент данных y типа int.
3. Выполняется тело конструктора класса Point.

Перед выполнением конструктора класса Rectangle дважды вызывается конструктор класса Point: в первый раз для инициализации полей элемента данных pt1, во второй — для инициализации полей элемента данных pt2.

Заметим, что при уничтожении составного объекта процесс управления памятью и вызова функций происходит в обратном порядке. Сначала, до освобождения какой-либо памяти, вызывается деструктор составного класса. Когда этот деструктор завершает работу, элементы данных уничтожаются в порядке, обратном их созданию. Уничтожение каждого элемента данных выполняется рекурсивно. Перед освобождением памяти компонента вызывается его деструктор. После завершения деструктора компонента уничтожаются элементы данных (от последнего в спецификации компонентного класса к первому).

Таким образом, при уничтожении объекта Rectangle последовательность событий будет зеркальным отражением событий, имевших место при его создании.

1. Выполняется деструктор класса Rectangle.
2. Уничтожается элемент данных thickness.
3. Для элемента данных pt2 вызывается деструктор Point.
4. Уничтожается элемент данных pt2 (сначала поле y, потом поле x).
5. Для элемента данных pt1 вызывается деструктор Point.
6. Уничтожается элемент данных pt1 (сначала поле y, потом поле x).

При описании процесса создания объекта Rectangle можно было с уверенностью говорить о вызове конструктора Rectangle, поскольку класс Rectangle имеет только один конструктор. Когда речь идет о выполнении конструктора класса Point, такой уверенности нет. Какой конструктор вызывается при создании полей объекта класса Point?

Как и в случае функции C++, ответ зависит от числа аргументов, подставляемых клиентом в вызове конструктора. Концептуальным "камнем преткновения" для многих программистов является то, что при отсутствии аргументов они не видят, какой именно конструктор вызывается. Между тем, когда нет аргументов, вызывается конструктор без аргументов, т. е. конструктор по умолчанию. Если указывается один аргумент, то задается конструктор с аргументом данного типа и т. д. А что бывает в случае, если конструктор с требуемой сигнатурой недоступен? Подобно вызову любой функции C++ с некорректной сигнатурой, это означает, что вызов функции (попытка создания объекта) даст синтаксическую ошибку.

В следующем фрагменте клиента можно видеть, что программа передает параметры для вызова конструктора Rectangle, однако здесь нет параметров, которые отправляли бы данные конструкторам Point.

```
Point p1(20,40), p2(70,90);          // верхний левый и нижний правый углы
Rectangle rec(p1,p2,4);              // это синтаксическая ошибка
```

Это означает, что когда в процессе создания составного объекта задается элемент данных компонентного типа, вызывается используемый по умолчанию конструктор компонентного класса.

В нашем примере класс Point не имеет конструктора по умолчанию. Вспомним, однако, что в этом случае конструктор по умолчанию для класса подставляет компилятор. Данный конструктор ничего не делает, но позволяет клиенту создавать объекты, не передавая аргументы конструктору. И это хорошие новости. Теперь вспомним, что если класс имеет отличные от используемых по умолчанию конструкторы (у класса Point есть один такой конструктор), то компилятор не подставляет свой конструктор. Это плохие новости. Определение объекта составного класса даст синтаксическую ошибку. Вот почему последняя строка в приведенном выше примере ошибочна.

Данный пример показывает связь между классами (Point и Rectangle), которая не очевидна для программистов. Некоторые полагают, что ошибка в определении класса Rectangle вызвана его попыткой создать несуществующий конструктор класса Point. Между тем компиляция класса Point и класса Rectangle не дает синтаксической ошибки.

Рекомендуется искать источник ошибки в архитектуре компонентного класса Point. Именно этот класс не имеет конструктора по умолчанию. Однако логическая ошибка проявляется как синтаксическая не в архитектурах компонентного класса Point и составного класса Rectangle, а в клиенте класса Rectangle при попытке создания экземпляра объекта составного класса. Пока клиент не попытается определить объект Rectangle, никакой синтаксической ошибки не будет.

Чтобы исправить ситуацию, можно добавить к классу Point конструктор по умолчанию. Это устранит синтаксическую ошибку в последнем фрагменте.

```
class Point {
    int x, y;                        // закрытые координаты
  public:
    Point ()
       { x=0; y=0; }                 // конструктор по умолчанию
    Point (int a, int b)             // обобщенный конструктор
       { x = a; y = b; }
  . . . } ;                          // остальная часть класса Point
```

Еще одно решение состоит в добавлении в обобщенный конструктор значений аргументов по умолчанию. Таким образом, он будет представлять собой используемый по умолчанию конструктор и конструктор преобразования.

```
class Point {
  int x, y;                          // закрытые координаты
public:
  Point (int a=0, int b=0)
  { x=a; y=b; }                      // конструктор по умолчанию, конструктор
                                     // преобразования, обобщенный конструктор
  . . . } ;                          // остальная часть класса Point
```

Проанализируем шаги создания объекта Rectangle. На рис. 12.1 показаны действия при выполнении следующего фрагмента:

```
Point p1(20,40), p2(70,90);     // верхний левый и нижний правый углы
Rectangle rec(p1,p2,4);         // OK, если Point имеет конструктор по умолчанию
```

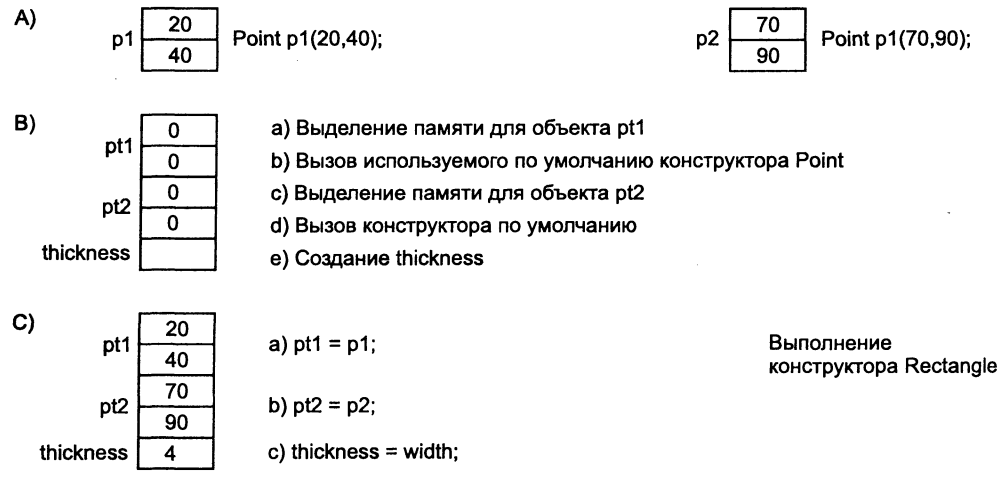


Рис. 12.1. *Шаги создания объекта Rectangle с помощью вызовов конструктора по умолчанию*

Для объекта pt1 выделяется память. Вызывается конструктор по умолчанию Point, устанавливающий pt1.x и pt1.y в 0. Далее выделяется память для объекта pt2 и вызывается конструктор Point, устанавливающий pt2.x и pt2.y в 0. После этого выделяется и остается неинициализированной память для элемента данных thickness. Затем вызывается конструктор Rectangle. Когда выполняется тело конструктора, содержимое аргумента p1 сначала копируется в элемент данных pt1, затем конструктор копирует p2 и pt2, после этого thickness устанавливается в значение 4.

В результате данной последовательности событий создается объект Rectangle, при этом pt1.x устанавливается в 20, pt1.y — в 40, pf2.x — в 70, pt2.y — в 90.

Значения, помещенные в pt1 и pt2 конструктором Point, оказались недолговечными. Они были перезаписаны данными из p1 и p2 при вызове конструктора Rectangle. Два конструктора по умолчанию для двух элементов данных отработали напрасно. Какие чувства это у вас вызывает? Негодование? Сожаление по поводу напрасно потраченных при создании объекта Rectangle миллисекунд? Хорошо. Теперь вы знаете, что такое неверное программирование на C++.



Внимание Создание экземпляров объектов в C++ всегда влечет за собой вызов функции — конструктора класса. При задании составных объектов в C++ вызывается несколько конструкторов. Конструктор вызывается сразу после создания каждого элемента данных. Нужно научиться видеть эти вызовы конструкторов в любой программе C++.

```
Создан: x=20 y=40
Создан: x=70 y=90
Создан: x=100 y=120
Создан: x=0 y=0
Создан: x=0 y=0
Присвоен: x=20 y=40
Присвоен: x=70 y=90
Точка внутри
```

Рис. 12.2.
Результат выполнения программы из листинга 12.1

Программист, работающий на языке C++, не должен напрасно тратить время при выполнении программы. Он должен научиться видеть лишние вызовы в любой программе C++ и знать, как избежать их там, где это возможно.

Листинг 12.1 показывает реализацию составного класса Rectangle и класса-компонента Point с тестовой программой. Чтобы облегчить анализ результатов, к классу Point добавлены конструктор копирования и перегруженная операция присваивания с отладочными сообщениями, позволяющими отследить процесс создания составного объекта Rectangle. Результат выполнения данной программы представлен на рис. 12.2.

Листинг 12.1. Пример создания составного объекта с лишними вызовами конструкторов

```
#include <iostream>
using namespace std;

class Point {
  private:
    int x, y;                                    // закрытые координаты
  public:
    Point (int a=0, b=0)                         // общий конструктор
    { x = a; y = b;
      cout << "Создан: x= " << x << " y=" << y << endl; }
    Point (const Point& pt)                      // конструктор копирования
    { x = pt.x; y = pt.y;
      cout << " Скопирован: x= " << x << " y=" << y << endl; }
    void operator = (const Point& pt)            // операция присваивания
    { x = pt.x; y = pt.y;
      cout << " Присвоен: x= " << x << " y=" << y << endl; }
    void set (int a, int b)                      // функция-модификатор
      { x += a; y += b; }
    void move (int a, int b)                     // функция-модификатор
      { x += a; y += b; }
    void get (int& a, int& b) const              // функция-селектор
      { a = x; b = y; } } ;

class Rectangle {
  Point pt1, pt2;                                // верхний левый и нижний правый угол
  int thickness                                  // толщина границы прямоугольника
public:
  Rectangle (const Point& p1, const Point& p2, int width=1);
  void move(int a, int b);                       // перемещение обеих точек
  bool pointIn(const Point& pt) const;           // точка в прямоугольнике?
  } ;

Rectangle::Rectangle(const Point& p1, const Point p2, int width)
  { pt1 = p1; pt2 = p2; thickness = width; }     // установка значений данных

void Rectangle::move(int a, int b)
  { pt1.move(a,b); pt2.move(a,b); }              // передача работы в Point

bool Rectangle::pointIn(const Point& pt) const   // точка внутри?
{ int x,y,x1,y1,x2,y2;                           // координаты pt углов
  pt.get(x,y);                                   // получение координат параметра
  pt1.get(x1,y1); pt2.get(x2,y2);                // получение данных угловых точек
  bool xIsBetweenBorders = (x1<x && x<x2 || (x2<x && x<x1);
  bool yIsBetweenBorders = (y>y1 && y<y2) || (y<y1 && y>y2);
  return (xIsBetweenBorders && yIsBetweenBorders); }

int main()
{
  Point p1(20,40), p2(70,90);                    // левый верхний, правый нижний углы
  Point point(100,120);                          // точка для отлова в прямоугольнике
  Rectangle rec(p1,p2,4);                        // напрасные вызовы конструктора
  point.move(-25,-15);                           // перемещение точки по экрану
  rec.move(10,20);                               // на 10 пикселей вправо, на 20 - вниз
  if (rec.pointIn(point)) cout << " Точка внутри\n";          // точка внутри?
  return 0;
  }
```

Первые три сообщения "Создан" на рис. 12.2 отражают создание объектов p1 и p2 типа Point и объекта Point. Следующие два сообщения "Создан" означают создание объекта Rectangle: первое описывает создание элементов данных pt1 и вызов конструктора по умолчанию Point, а второе — создание элемента данных pt2 и вызов используемого по умолчанию конструктора Point. Два сообщения "Присвоен" описывают выполнение конструктора Rectangle после завершения создания объекта. Первое сообщение соответствует первому присваиванию в теле конструктора, а второе — второму.

Это типичная картина создания составного объекта в C++. Для крупных составных объектов данный процесс может быть сложным и связанным с большим числом лишних вызовов конструкторов. Конечно, вы не сможете исключить вызовы конструкторов, активизируемые сразу после создания каждого элемента данных. В C++ не может быть создан объект, за которым сразу не следует вызов конструктора. Однако надо стараться вызывать такой конструктор, который не будет действовать после выполнения конструктора составного объекта.

Использование списка инициализации элементов

Язык C++ позволяет использовать *список инициализации элементов* в конструкторе составного класса. В нем применяется не вполне обычный синтаксис, основанный на включении списка между заголовком и телом конструктора. Вот как выглядит инициализация элементов:

```
class Rectangle {
  Point pt1, pt2;                          // верхний левый, нижний правый
  int thickness;
public:
  Rectangle (const Point& p1, const Point& p2, int w = 1);
  . . . . } ;                              // остальная часть класса Rectangle

Rectangle::Rectangle(const Point& p1,
              const Point& p2, int w) : pt1(p1),pt2(p2)
  { thickness = w; }                       // это намного лучше!
```

Список инициализации помещается между закрывающей круглой скобкой списка параметров конструктора и открывающей фигурной скобкой тела конструктора. Он открывается двоеточием и перечисляет имена (а не типы) элементов данных. После каждого имени элемента в круглых скобках указывается соответствующее значение (значения), используемое для инициализации данного объекта — элемента данных. Список не имеет терминатора. Он заканчивается открывающей фигурной скобкой тела класса. Каждая запись в списке аналогична вызову конструктора в определении переменной, например pt1(p1).

Заметим, что список инициализации элементов применяется только к реализации конструктора. Не путайте список инициализации со значениями параметров по умолчанию. Синтаксис значений по умолчанию применяется только к прототипам и не влияет на способ записи реализации конструктора.

Список инициализации вынуждает компилятор генерировать вызов конструктора элементов данных с соответствующим числом параметров. Этот конструктор вызывается после выделения памяти для элемента данных и перед телом конструктора составного класса. Следовательно, элементы данных ко времени вызова конструктора составного класса уже инициализированы. При необходимости их можно использовать в конструкторе составного класса.

Фактически любые элементы данных могут инициализироваться в списке. Конструктор класса Rectangle, где все элементы данных инициализируются в списке:

```
Rectangle::Rectangle(const Point& p1, const Point& p2, int w)
    : thickness(w), pt1(p1), pt2(p2)        // на отдельной строке
   { }                                  // пустое тело: популярная идиома в C++
```

Как видно из примера, список инициализации располагается на отдельной строке. Это распространенный вариант синтаксиса. При использовании такого расширенного инициализатора возникает непростая ситуация. К моменту выполнения тела конструктора делать уже ничего не нужно. Вот почему тело конструктора здесь пустое. Однако оно все равно должно присутствовать, поскольку тело функции опускать нельзя. Инициализация отдельных элементов данных в списке инициализации не дает никаких преимуществ, но по некоторым причинам пустое тело конструктора — популярный прием среди программистов, использующих язык C++.

Список инициализации может создать впечатление, что элемент данных thickness инициализируется перед pt1 и pt2. Это не так. Не важно, в каком порядке следуют элементы в списке инициализации. Они получают значение в обратном порядке их следования в спецификации класса. Это тот случай, когда внешность обманчива.

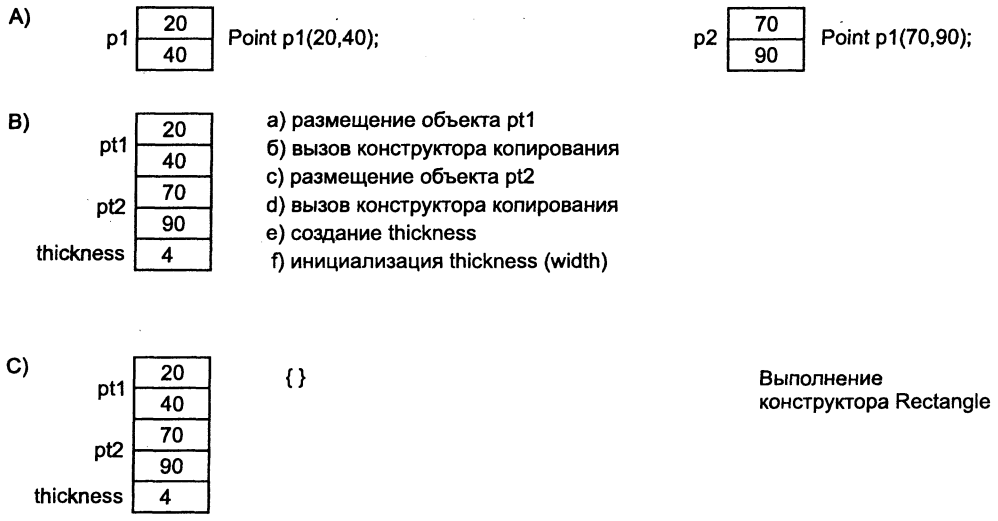


Рис. 12.3. *Шаги создания объекта Rectangle со списком инициализации элементов*

На рис. 12.3 показана последовательность событий, происходящих при создании объекта со списком инициализации в данном примере.

```
Point p1(20,40), p2(70,90);          // верхний левый и нижний правый углы
Rectangle rec(p1,p2,4);              // не нужен конструктор Point по умолчанию
```

После выделения памяти для p1 и p2 создается объект rec типа Rectangle. Сначала задается элемент данных pt1, затем для него объектом p1 в аргументе вызывается конструктор копирования Point. В результате элемент данных pt1 инициализируется: x равно 20, y — 40. Далее создается элемент данных pt2. Для него вызывается конструктор копирования Point с передачей объекта p2 в аргументе. В результате инициализируется элемент данных pt2: x равно 70, y — 90. После этого создается и инициализируется значением 4 элемент данных thickness. Лишние вызовы используемого по умолчанию конструктора класса Point исчезли.

```
Создан: x=20 y=40
Создан: x=70 y=90
Создан: x=100 y=120
Скопирован: x=20 y=40
Скопирован: x=70 y=90
Точка внутри
```

Рис. 12.4. *Результат выполнения программы из листинга 12.2*

В листинге 12.2 показана такая же программа, что и в листинге 12.1, но с измененной архитектурой класса Rectangle. Однако эта программа реализует конструктор Rectangle со списком инициализации элементов. Результат выполнения данной программы показан на рис. 12.4.

Листинг 12.2. Создание составного объекта без лишних вызовов конструкторов

```
#include <iostream>
using namespace std;

class Point {
  private:
    int x, y;                                    // закрытые координаты
  public:
    Point (int a=0, b=0)                         // общий конструктор
    { x = a; y = b;
      cout << "Создан: x= " << x << " y=" << y << endl; }
    Point (const Point& pt)                      // конструктор копирования
    { x = pt.x; y = pt.y;
      cout << "Скопирован: x= " << x << " y=" << y << endl; }
    void operator = (const Point& pt)            // операция присваивания
    { x = pt.x; y = pt.y;
      cout << "Присвоен: x= " << x << " y=" << y << endl; }
    void set (int a, int b)                      // функция-модификатор
      { x = a; y = b; }
    void move (int a, int b)                     // функция-модификатор
      { x +=a; y +=b; }
    void get (int& a, int& b) const              // функция-селектор
      { a = x; b = y; } } ;

class Rectangle {
  Point pt1, pt2;                                // верхний левый и нижний правый угол
  int thickness                                  // толщина границы прямоугольника
public:
  Rectangle (const Point& p1, const Point& p2, int width=1);
  void move(int a, int b);                       // перемещение обеих точек
  bool pointIn(const Point& pt) const;           // точка в прямоугольнике?
    } ;

Rectangle::Rectangle(const Point& p1, const Point& p2, int w)
: thickness(w), pt1(p1), pt2(p2)                 // список инициализации
  { }                                            // пустое тело конструктора

void Rectangle::move(int a, int b)
  { pt1.move(a,b); pt2.move(a,b); }              // передача работы в Point

bool Rectangle::pointIn(const Point& pt) const   // точка внутри?
  {  int x,y,x1,y1,x2,y2;                        // координаты pt углов
     pt.get(x,y);                                // получение координат параметра
     pt1.get(x1,y1); pt2.get(x2,y2);             // получение данных угловых точек
     bool xIsBetweenBorders = (x1<x && x<x2 || (x2<x && x<x1);
     bool yIsBetweenBorders = (y>y1 && y<y2) || (y<y1 && y>y2);
     return (xIsBetweenBorders && yIsBetweenBorders); }

int main()
{  Point p1(20,40), p2(70,90);                   // левый верхний, правый нижний углы
   Point point(100,120);                         // точка для отлова в прямоугольнике
   Rectangle rec(p1,p2,4);                       // напрасные вызовы конструктора
   point.move(-25,-15);                          // перемещение точки по экрану
   rec.move(10,20);                              // на 10 пикселей вправо, на 20 - вниз
   if (rec.pointIn(point)) cout << " Точка внутри\n";          // точка внутри?
   return 0;
   }
```

Первые три сообщения "Создан" на рис. 12.4 соответствуют первым трем сообщениям на рис. 12.2. Они показывают процесс создания трех объектов Point в функции main(). Следующие два сообщения "Скопирован" демонстрируют процесс создания объекта Rectangle. Первое сообщение появляется, когда после создания элемента данных pt1 вызывается конструктор копирования. Второе сообщение выводится, когда конструктор копирования Point задается после создания элемента данных pt1. Как видно, вызывается именно конструктор копирования Point, а не конструктор по умолчанию. В результате операция присваивания Point не вызывается при вызове конструктора Rectangle. Тело этого конструктора пустое.

В этих примерах составной класс имел только один конструктор. Если бы конструкторов было несколько, каждый из них создавался бы одинаково. При создании составного объекта сначала задаются его элементы данных. Какой именно конструктор класса вызывается в конце, зависит от числа и типов аргументов, подставляемых клиентом для составного объекта. Если составной класс имеет такой конструктор — замечательно. Если же нет, то при инициализации объекта будет синтаксическая ошибка. Когда вызываемый в итоге конструктор не имеет списка инициализации элементов, за созданием каждого из них следует вызов используемого по умолчанию конструктора класса, к которому принадлежит элемент. Если конструктор составного класса имеет список инициализации, для каждого элемента в нем вызывается соответствующий конструктор класса компонента.

Списки инициализации, реализуемые разными конструкторами, могут различаться. Вот еще одна версия класса Rectangle, совмещающая три конструктора: общий конструктор, использовавшийся в предыдущих примерах, общий конструктор с четырьмя параметрами (координатами двух точек) и конструктор по умолчанию. Каждый конструктор имеет свой собственный список инициализации. Эти списки не обязаны быть одинаковыми.

```
class Rectangle {
  Point pt1, pt2;                          // верхний левый, нижний правый
  int thickness;
public:
  Rectangle (const Point& p1, const Point& p2, int w = 1);
  Rectangle (int x1, int y1, int x2, int y2);
  Rectangle ();
  . . . . } ;                              // остальная часть класса Rectangle

Rectangle::Rectangle(const Point& p1, const Point& p2, int w)
  : thickness(w), pt1(p1), pt2(p2) { }

Rectangle::Rectangle (int x1, int y1, int x2, int y2)
  : pt1(x1,y1), pt2(x2,y2), thickness (1) { }

Rectangle::Rectangle () : pt1(0,0), pt2(100,100), thickness(1)
  { }
```

Первый конструктор демонстрирует, как можно передавать параметры конструктору составного класса для подстановки параметров конструктора класса компонента. В этом примере параметр p1 типа Point перенаправляется как аргумент конструктору копирования для инициализации элемента данных pt1, а параметр p2 типа Point передается как аргумент конструктору копирования для инициализации элемента данных pt2. Последний параметр конструктора применяется для инициализации элемента данных типа int.

Второй конструктор показывает, что список инициализации элементов данных не ограничивается использованием параметров, подставляемых клиентом. Здесь клиент подставляет лишь данные, которые используются для вызова общего конструктора Point. Значение, инициализирующее элемент данных thickness, определяется как литеральная константа.

Обратите внимание, что применение литеральных значений в списке инициализации представляет собой вид переноса обязанностей с клиента на серверный класс. В данной версии программист класса Rectangle определяет толщину линии, равную единице. Можно заменить литеральное значение дополнительным параметром, как в следующей версии конструктора:

```
Rectangle::Rectangle (int x1, int y1, int x2, int y2), int width)
  : pt1 (x1,y1), pt2(x2,y2), thickness (width) { }
```

В таком случае задавать толщину линии, равную единице, должен клиент.

```
Rectangle r(20,40,70,90,1);              // обязанности переносятся на клиента
```

Третий конструктор класса Rectangle, конструктор по умолчанию, также демонстрирует использование литералов в списке аргументов. При применении этого конструктора каждый объект Rectangle инициализируется таким образом, что его верхним левым углом будет точка с координатами (100,100). Конструктор по умолчанию не получает от клиента никаких данных. Таким образом, объекты, инициализируемые используемым по умолчанию конструктором, устанавливаются в одинаковое состояние.

Класс Point предусматривает конструктор по умолчанию (с нулевыми значениями координат), поэтому список инициализации Rectangle для него можно записать следующим образом:

```
Rectangle::Rectangle () : pt1(), pt2(100,100), thickness(1) { }
```

Что произойдет, если вы пропустите список инициализации для элементов данных pt1? Вспомните: назначение списка инициализации элементов состоит в том, чтобы избежать вызова используемого по умолчанию конструктора класса компонента перед вызовом конструктора составного класса. Список инициализации элементов заменяет вызов применяемого по умолчанию конструктора класса компонента на вызов конструктора, заданного в списке инициализации.

Таким образом, в данном примере конструктора Rectangle вы пытаетесь избежать вызова для компонента pt1 используемого по умолчанию конструктора Point и заменить его на вызов конструктора по умолчанию для компонента pt1! Следовательно, вызовы конструкторов по умолчанию в списке инициализации компонентов можно опустить. Это не повлияет на последовательность событий при создании объекта составного класса. Последнюю версию конструктора Rectangle можно переписать так:

```
Rectangle::Rectangle () : pt2(100,100), thickness(1) { }        // то же самое
```

Синтаксис списка инициализации выглядит странно. Похоже, ничего подобного вам раньше не встречалось. У многих программистов при изучении такого синтаксиса возникают сложности.



Советуем Изучите синтаксис списка инициализации. Это очень полезно и позволяет инициализировать компоненты составных объектов без лишних вызовов конструкторов. Такой метод весьма популярен в C++. Позднее будут приведены примеры его использования в наследовании. Чем скорее вы его изучите, тем лучше для вас.

Вы можете изучать этот синтаксис постепенно. Во-первых, вам надо избежать синтаксических ошибок, когда отсутствует используемый по умолчанию конструктор компонента. Во-вторых, вам необходимо избежать отрицательного влияния на производительность, когда в конструкторе составного класса состояние компонента сбрасывается. Таким образом, целью является предотвращение синтаксических ошибок и повышение производительности. Синтаксических ошибок можно избежать, используя в классе компонента конструктор по умолчанию. Для констант и ссылочных элементов данных список инициализации обязателен.

Элементы данных со специальными свойствами

Возможно, прежде вы никогда не думали об использовании констант и ссылочных элементов данных. Когда в конце главы 8 перечислялись вопросы, которые удается разрешить в C++ с помощью классов, говорилось о связывании данных и операций, введении области действия класса для имен элементов данных и функций-членов, управлении доступом к компонентам классов, переносе обязанностей с клиентов на серверы.

Все это направлено на поддержку принципов объектно-ориентированного программирования. В главах 9–11 говорилось о других целях применения классов, которые также стали частью программирования: автоматическая инициализация объектов, управление динамической памятью, одинаковая интерпретация объектов и переменных встроенных типов. Теперь перейдем к определению элементов данных как констант или ссылок.

Константы как элементы данных

Идея констант или постоянных элементов данных проста. В классе C++ увязываются вместе элементы данных и функции. Функции реализуют доступ к элементам данных от лица клиента. Часто клиенту требуется изменить состояние объекта (например, сумму на счете, адрес сотрудника, цену за прокат видеофильма и т. д.). Однако некоторые характеристики объекта не предусматривают изменений (например, номер счета, дата рождения сотрудника, сумма, выплаченная за прокат видеофильма).

Разработчик класса знает, что функции-члены класса не изменяют отдельные элементы данных. Программисту, сопровождающему приложение, придется понять это, изучив функции-члены и "дружественные" функции класса. Было бы неплохо явно указать замыслы разработчика в классе: значение этого элемента данных не изменяется ни одной функцией-членом или "дружественной" функцией. Вот еще одна работа для ключевого слова `const`.

Как инициализировать такой компонент-константу? Делать это в конструкторе класса, вероятно, слишком поздно. Вспомните схему на рис. 12.1. Конструктор класса вызывается только после того, как объект уже создан. Для компонентов-констант присваивание не допускается. Постоянный элемент данных должен инициализироваться немедленно после создания этого компонента, поэтому C++ требует включать имя компонента-константы в список инициализации. Любое присваивание этому компоненту он помечает как синтаксическую ошибку.

Постоянный элемент данных может иметь тип, определяемый программистом, или встроенный тип. Это не важно для инициализации после создания элементов данных, перед вызовом конструктора.

В качестве примера применения компонента-константы добавим к классу `Rectangle` дополнительный элемент данных, показывающий вес единицы объема прямоугольника. Материал, из которого изготовлен прямоугольник, не меняется за время существования прямоугольника, и этот элемент данных не будет модифицироваться после создания объекта `Rectangle`. Чтобы сопровождающий приложение программист не перебирал в подтверждение данного факта все функции-члены и "дружественные" функции класса, элемент данных `weight` надо определить как константу. Следовательно, его необходимо инициализировать в списке инициализации конструктора `Rectangle`.

```
class Rectangle {
  Point pt1;
  Point pt2;
  int thickness;
  const double weight;                    // вес единицы объема многоугольника
```

```
public:
  Rectangle (const Point& p1, const Point& p2, double wt, int width = 1);
  void move(int a, int b);
  void setThickness(int w=1);
  int pointIn(const Point& pt) const;
  . . . . } ;                                  // остальная часть класса Rectangle

Rectangle::Rectangle(const Point& p1, const Point& p2, double wt, int width)
    : pt1(p1), pt2(p2), weight(wt)                  // вес здесь не обязателен
{ thickness = width; }
```

Обратите внимание, что дополнительный параметр добавлен не в конце списка параметров конструктора, а в середине. Вспомните правило, по которому значения по умолчанию допускаются только для самых правых параметров. Если добавить четвертый параметр в конец списка (как в следующей строке), это правило нарушается.

```
Rectangle (const Point& p1, const Point& p2, int width = 1, double wt);
                                                                    // неверно
```

Для клиента Rectangle не потребуется существенных изменений.

```
Point p1(20,40), p2(70,90);        // левый верхний, правый нижний углы
Point point(100,120);              // точка для отлова в прямоугольнике
Rectangle rec(p1,p2,0.01,4);       // напрасные вызовы конструктора
rec.setThickness();                // толщина линии - 1 пиксель (по умолчанию)
point.move(-25,-15);               // перемещение точки по экрану
rec.move(10,20);                   // на 10 пикселей вправо, на 20 - вниз
if (rec.pointIn(point)) cout << "Точка внутри\n";       // точка внутри?
p1.move(30,35);                    // изменяется ли объект Rectangle?
```

Подобно другим случаям использования ключевого слова const, применение постоянных элементов данных необходимо, чтобы сделать исходный код более понятным. Если элементы данных не имеют ключевого слова const, это должно означать их модификацию во время существования объекта одной из связанных с классом функций. Похоже, что лишь немногие программисты придерживаются правил использования компонентов-констант. Следовательно, отсутствие ключевого слова const в определении элементов данных не свидетельствует об их изменении. Это может говорить лишь о том, что программист был занят другими аспектами программы и не утруждал себя сообщением информации о своих замыслах.

Ссылочные элементы данных

Рассмотрим пример применения ссылок на объекты, используемые как элементы данных других объектов. Это хорошая программная реализация связи между объектами, где несколько клиентских объектов ассоциируются с одним серверным объектом.

В предыдущих примерах данной главы каждый объект Rectangle имел собственную копию своих угловых точек. Если перемещается p1 (как в последней строке предыдущего примера), то объект rec типа Rectangle не изменяется. Во многих приложениях объекты так себя и ведут. В главе 11 был представлен подобный подход с использованием семантики значений.

В других приложениях желательно совместно использовать угловые точки прямоугольников. Таким образом, если клиент перемещает угловую точку прямоугольника, он должен изменяться. В главе 11 такой подход поддерживался с помощью семантики ссылок. Во многих программах, разработанных с применением

объектно-ориентированной методологии, семантика ссылок служит для реализации связей между объектами. Например, данные о владельце счета (фамилия, адрес, номер социального страхования и т. д.) могут быть частью класса Account (счет). Если в приложении необходим класс Owner (владелец), эти данные можно скомбинировать в класс и использовать объект Owner как элемент данных класса Account. Если владелец имеет несколько счетов, то в приложении можно использовать для них только один объект Owner. Тогда изменения в данных владельца автоматически распространяются на все счета.

Для поддержки такой функциональности в клиенте можно применять ссылочные элементы данных. Эти ссылки обозначают объекты вне составного объекта. Такие внешние объекты могут модифицироваться в клиенте без знания составного объекта.

Как уже говорилось выше, все ссылки в C++ — это константы. Они не могут изменяться после инициализации. Следовательно, подобно постоянным элементам данных, ссылочные элементы должны инициализироваться только в своем списке. Никакая инициализация в теле конструктора не допускается. Там ее делать слишком поздно, так как конструктор вызывается после создания всех элементов данных и вызовов их конструкторов. Новая архитектура класса Rectangle подобна предыдущей версии. Единственная разница состоит в двух знаках амперсанда после типа Point в определениях элементов данных.

```
class Rectangle {
    Point& pt1;                          // точки могут использоваться
    Point& pt2;                          // совместно с другими фигурами
    int thickness;
    const int weight;                    // вес единицы объема многоугольника
  public:
    Rectangle (const Point& p1, const Point& p2,
                    int wid = 1, int wt = 1);
    void move(int a, int b);
    void setThickness(int w=1);
    int point(const Point &pt) const;
    . . . . } ;                          // остальная часть класса Rectangle

Rectangle::Rectangle(const Point& p1, const Point& p2, int width, int wt)
    : pt1(p1), pt2(p2), weight(wt)       // здесь это не обязательно
{ thickness = width; }                   // тот же конструктор, что и выше
```

Все ссылки — это константы, поэтому данное свойство специально обозначать не нужно. В классе Rectangle компоненты pt1 и pt2 являются ссылками-константами. Их нельзя "отсоединить" от объектов, на которые они ссылаются, и переназначить на другие объекты, однако сами объекты константами не являются. Их содержимое может изменяться.

Подобно объектам, на которые ссылаются указатели, объекты по ссылкам также могут определяться как константы. Это означает, что pt1 и pt2 не только обозначают одни и те же объекты Point и не могут переключаться на другие объекты Point, но и сами эти объекты не меняются и сохраняют свое состояние.

```
class Rectangle {
  const Point& pt1;                  // точки могут использоваться
                                     // совместно с другими фигурами
  const Point& pt2;                  // точки не могут изменять свои координаты
  . . . . } ;                        // остальная часть класса Rectangle
```

С синтаксической точки зрения эти требования одинаковы — элементы данных должны инициализироваться в списке инициализации. С семантической точки зрения смысла в подобной архитектуре немного. Если угловые точки — константы,

то их совместное использование с другими объектами Rectangle не дает никаких преимуществ. Их можно сделать компонентами-константами.

```
class Rectangle {
  const Point& pt1;              // точки не используются
                                 // совместно с другими фигурами
  const Point& pt2;              // точки не могут изменять свои координаты
  . . . . } ;                    // остальная часть класса Rectangle
```

Ссылку на объект-константу можно применять как метод оптимизации. Если большое количество составных объектов должно содержать один и тот же объект-компонент, имеет смысл создать только один такой объект, а во всех составных объектах использовать ссылки на данный объект.

Процесс построения объекта класса Rectangle представлен на рис. 12.5. Сначала создаются объекты типа Point (см. рис. 12.5(A)), затем — объект Rectangle (см. рис. 12.5(B)): задаются ссылки pt1 и pt2. Их тип отличается от типа в предыдущем примере. Используются другие размеры фигур, чтобы показать разницу типов. Выполняется список инициализации, и ссылки устанавливаются на объекты Point, создается и инициализируется поле-константа weight, а затем — файл thickness. Вызывается конструктор Rectangle (см. рис. 12.5(C)). Присваивается значение поля thickness.

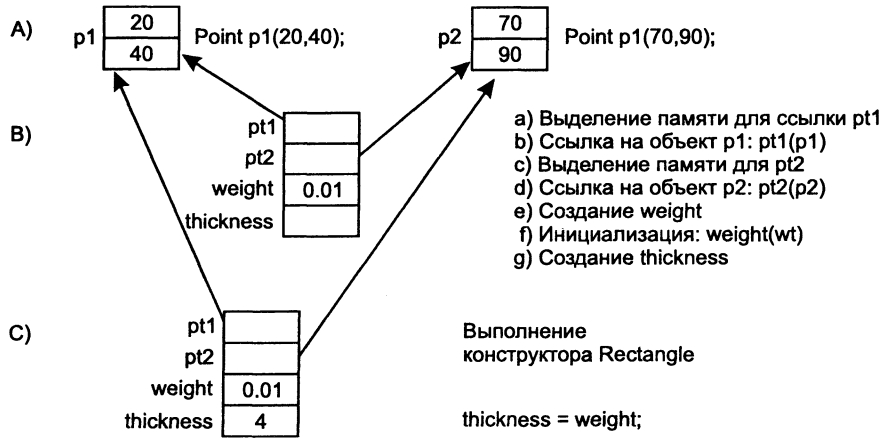


Рис. 12.5. *Этапы создания объекта Rectangle со ссылками на внешние объекты Point*

В данной архитектуре ссылки в классе Rectangle представляют собой константы, но объекты класса Point, на которые они указывают, не являются константами. Следовательно, они могут менять свое состояние. Все объекты Rectangle, ассоциированные с этими объектами Point, могут изменять свое положение на экране. За счет ссылок объект Rectangle не может потерять связанный с ним объект Point и использовать вместо него другой объект Point. Это возможно, если класс Rectangle вместо ссылок применяет указатели.

```
class Rectangle {
  Point *pt1, *pt2;                  // совместно с другими фигурами
  int thickness;
  const int weight;                  // вес единицы объема многоугольника
public:
  Rectangle (const Point*, const Point*, int = 1, int = 1);
  void move(int a, int b);
```

```
    void setThickness(int w=1);
    int pointIn(const Point &pt) const;
    . . . . } ;                          // остальная часть класса Rectangle

Rectangle::Rectangle(const Point *p1, const Point *p2,
   int width, int wt) : pt1(p1), pt2(p2), weight(wt)   // тоже не обязательно
  { thickness = width; }                    // тот же конструктор, что и выше
```

Так как указатели могут изменяться в любой момент своего существования, применение списка инициализации здесь не обязательно, но это хорошая практика.

Если объекты, на которые ссылаются указатели, остаются постоянными в течение всего времени существования объекта Rectangle, их можно объявить как указатели на объекты-константы.

```
class Rectangle {
  const Point *pt1;              // точки могут использоваться
                                 // совместно с другими фигурами
  const Point *pt2;              // точки не могут изменять свои координаты
  . . . . } ;                    // остальная часть класса Rectangle
```

Такая конструкция может дать полезную оптимизацию, если большое число объектов Rectangle ассоциировано с одними и теми же объектами Point.

Не злоупотребляйте использованием ссылочных и постоянных элементов данных. Если жеони отражают свойства серверных объектов и потребности клиентов, их следует рассматривать как законные и полезные инструменты для разработки программы.

Использование объектов как элементов данных своего собственного класса

В предыдущих разделах обсуждалась ситуация, когда объект одного класса (например, Point) использовался как элемент данных другого класса (например, Rectangle). Может ли объект класса быть компонентом своего собственного класса? Например, приложению могут потребоваться координаты точек, относящихся к определенным зонам экрана, и спецификация точки привязки как характеристики класса Point.

```
class Point {
    int x, y;                                  // закрытые координаты
    Point anchor;                              // это не допускается
  public:
    Point (int a=0, int b=0)                   // многосторонний конструктор
          { x = a; y = b; }
. . . . } ;                                    // остальная часть класса Point
```

Между тем это не допускается. В начале главы уже Рассматривалась последовательность событий при распределении памяти для объекта. Память для элементов данных выделяется в порядке их определения в спецификации класса. (Для статических компонентов — в начале выполнения программы.) Таким образом, при создании объекта Point сначала выделяется память для x и y, а затем — для anchor ("якоря", точки привязки). Однако anchor имеет тип Point, поэтому вновь определяется память для его компонентов x, y, а затем anchor и т. д. Такой рекурсивный процесс не заканчивается, поэтому он запрещен.

Ссылки на объекты собственного класса допустимы (как, например, обозначения объектов одного и того же класса). И указатели, и ссылки представляют адрес объекта и не требуют распределения памяти для всего объекта. Следовательно,

память для них выделяется наряду с другими элементами данных без дополнительных трудностей.

```
class Point {
  int x, y;                          // закрытые координаты
  Point &anchor;                     // это допускается
public:
  Point (int a=0, int b=0, Point &focus)
    : anchor (focus)                 // не может устанавливаться в конструкторе
      { x = a; y = b; }
. . . } ;                            // остальная часть класса Point
```

Как уже говорилось в предыдущем разделе, ссылочные элементы данных не могут инициализироваться в теле конструктора класса, поэтому здесь они инициализируются в списке с помощью объекта класса Point, передаваемого конструктору в виде аргумента. Параметр конструктора не имеет модификатора const, потому что элемент данных anchor не определен как константа. Через ссылку anchor объект Point может модифицировать объект, переданный как аргумент. Вот почему определение параметра как константы было бы синтаксической ошибкой.

Обратите внимание, что здесь снова присутствует распространенная ошибка: новый параметр добавляется в конструкторе справа. У него нет значения по умолчанию, следовательно, его нужно перенести влево от параметров, имеющих такие значения.

```
class Point {
    int x, y;                                  // закрытые координаты
    Point &anchor;                             // это допускается
  public:
    Point (Point &focus, int a=0, int b=0)     // правильный порядок
      : anchor (focus)                         // не может устанавливаться в конструкторе
        { x = a; y = b; }
. . . } ;                                      // остальная часть класса Point
```

Клиент сначала создает базовую точку (anchor), а затем передает ее в аргументе конструктору новых точек. Первая точка должна быть базовой.

```
Point p1(p1);          // синтаксическая ошибка: p1 не определяется как аргумент
```

Это синтаксическая ошибка. Когда p1 используется в аргументе конструктора, данный объект создается. Следовательно, ссылка на него пока не определена. Поэтому память для базовой точки выделяется динамически.

```
Point *p = new Point(*p,80,90);          // p еще не имеет значения
Point p1(*p);                            // *p используется как базовая точка
```

Здесь возникает проблема: при использовании p как аргумента конструктора данная переменная еще не имеет значения, однако это приводит не к синтаксической ошибке, а к предупреждению. Избежать предупреждения можно путем инициализации указателя с помощью нулевого значения перед его использованием.

```
Point *p = 0;                    // важно избежать предупреждения
p = new Point(*p,80,90);         // динамическое распределение объекта Point
Point p1(*p);                    // используется как базовая точка
```

Как видно, применение ссылочных элементов данных на объект того же класса кажется хорошей идеей, но при этом из-за рекурсивности структуры данных возникает ряд трудностей. Не следует прибегать к такому приему, если в этом нет абсолютной необходимости.

Использование статических элементов данных как компонентов собственного класса

Использование статических элементов данных как компонентов собственного класса допускается, и это намного проще, чем применение ссылочных элементов данных. Например, объекты класса Point могут иметь общую для всех точек плоскости точку начала координат. Такая точка является общей, поэтому ее можно представить как статический элемент данных.

```
class Point {
    int x, y;                          // закрытые координаты
    static int count;
    static Point origin;               // статический объект, OK
  public:
    Point (int a, int b)
       { x = a; y = b; count++; }
. . . . } ;
```

Синтаксис инициализации статического объекта такой же, как для статических элементов данных встроенных типов. (Подробнее о статических элементах данных рассказывается в главе 9.) Следующая строка показывает пример определения и инициализации статического объекта. Первое Point здесь представляет тип определяемых элементов данных (origin). Второе показывает, что определяемый элемент данных принадлежит к классу Point. При создании объекта для инициализации его полей вызывается конструктор. В данном случае это обобщенный конструктор Point с двумя параметрами.

```
Point Point::origin(640,0);          // инициализация с помощью конструктора
```

Это аналогично определению статических элементов данных встроенных типов, например count. Данное поле имеет целый тип, и операция области действия показывает, что оно принадлежит к классу Point. Начальное значение поля устанавливается в нуль.

```
int Point::count = 0;
```

Подобно другим статическим объектам, не вполне понятно, когда именно создается объект и вызывается конструктор. Если в программе несколько статических переменных, то порядок их создания не определен. Расположение их в определенном порядке в исходном коде еще не гарантирует, что они будут создаваться и инициализироваться именно так. Отмечается лишь то, что они будут созданы перед выполнением первого оператора функции main().

В случае класса Point такой гарантии недостаточно. Конструктор Point вызывается как для нестатических объектов Point, так и для статического объекта origin, создаваемого первым. При этом статический элемент данных count должен быть создан перед объектом origin.

Статические элементы данных можно использовать в функции-члене своего класса (например, в конструкторе) как аргументы по умолчанию. Нестатические элементы данных для этого применять нельзя.

```
class Point {
  int x, y;
  static int count;
  static Point origin;                    // статический объект, OK
  Point &anchor;                          // ссылка или указатель, OK
public:
  Point (Point &focus = origin, int a=0, int b=0) : anchor(focus)
       { x = a; y = b; count++; }
```

```
void set (int a = x, int b)          // ошибка, нестатический элемент данных
{ x = a; y = b; }
. . . . } ;                          // остальная часть класса Point
```

Статические элементы данных создаются до начала выполнения программы. Доступ к ним возможен до создания объектов классов.

После создания объектов Point элементы данных count и origin могут быть доступны с помощью любого объекта. Результат будет одним и тем же, поскольку они статические. В отличие от нестатических элементов данных к ним нельзя обращаться с помощью имени класса, а не имени целевого объекта.

```
int main()
{ Point p1, p2(70,90);
  cout << "Число точек: " << p1.count << endl;          // выводит 2
  cout << "Число точек: " << p2.count << endl;          // выводит 2
  . . . }
```

Кроме того, в отличие от нестатических элементов данных, к ним нельзя обращаться с помощью имени класса с операцией области действия вместо имени целевого объекта с операцией-селектором.

```
int main()
{ Point p1, p2(70,90);
  cout << "Число точек: " << Point::count << endl;       // также выводит 2!
  . . . }
```

Более того, данный синтаксис доступен, даже когда объекты классов еще не созданы.

```
int main()
{ cout << "Число точек: " << Point::count << endl;      // выводит 0
  . . . }
```

Листинг 12.3 показывает вариант класса Point с двумя статическими элементами данных — count и origin. Их можно инициализировать вне определения класса, хотя они являются закрытыми. Функция quantity() определяется как статическая, и к ней можно обращаться с помощью операции области действия (первый вызов) и целевого объекта (второй вызов).

Листинг 12.3. Использование статических элементов данных и статической функции-члена

```
#include <iostream>
using namespace std;

class Point {
     int x, y;                                        // закрытые координаты
     static int count;
     static Point origin;
  public:
     Point (int a=0, int b=0)                         // обобщенный конструктор
       { x = a; y = b; count++;
         cout << Создан: x=" << x << " y=" << y
              << " count=" << count << endl; }
     static int quantity()                            // const не допускается
            { return count; }
     void set (int a, int b)                          // функция-модификатор
       { x = a; y = b; }
     void get (int& a, int& b) const                  // функция-селектор
       { a = x; b = y; }
```

```
    void move (int a, int b)                    // функция-модификатор
      { x += a; y += b; }
  ~Point()                                      // деструктор
    { count-;
      cout <<" Объект Point уничтожен: x=" <<x << "y=" <<y <<endl; }
  } ;
int Point::count = 0;                           // инициализация
Point Point::origin(640,0);                     // инициализация

int main()
{ cout << " Число точек: " << Point::quantity() << endl;
  Point p1, p2(30), p3(50,70);                  // точка начала координат, объекты точек
  cout << " Число точек: " << p1.quantity() << endl;
  return 0;
  }
```

```
Создан: x=640, y=0, count=1
Число точек: 1
Создан: x=0, y=0, count=2
Создан: x=30, y=0, count=3
Создан: x=50, y=70, count=4
Число точек: 4
Объект Point уничтожен: x=50  y=70
Объект Point уничтожен: x=30  y=0
Объект Point уничтожен: x=0  y=0
```

Рис. 12.6. *Вывод программы из листинга 12.3*

Результат данной программы представлен на рис. 12.6. Как видно, переменная count не равна нулю, хотя экземпляры объектов Point явно не создавались. Это подсчитан статический объект origin. Данный объект создан до выполнения первого оператора main(). Когда вызывается его конструктор, появляется отладочное сообщение "Создан" (первое в выводе). Этот статический элемент данных уничтожается после завершения программы, поэтому сообщение об исчезновении так и не появляется.

Если поменять определения переменных count и origin, результат программы не изменится.

```
Point Point::origin(640,0);
int Point::count = 0;
```

Это означает, что компилятор может отслеживать зависимость между двумя данными переменными и обеспечивать доступность переменной count во время выполнения конструктора Point для статического объекта origin.

В этом разделе шла речь о сложных методах программирования. Не следует применять их лишь потому, что вам нравятся трудные задачи. Пожалейте тех, кто будет работать с вашей программой.

Контейнерные классы

В предыдущем разделе обсуждались специальные случаи для элементов данных составных классов, которые, вероятно, вам не потребуется использовать ежедневно. В данном разделе описываются особые случаи для элементов данных составных классов, применяемые ежедневно. Даже если вам не придется самостоятельно писать подобные классы, наверняка надо будет работать с контейнерными классами, написанными другими программистами. *Контейнерные классы* — это особый вид составных классов. Советуем использовать их, когда приложению необходим тип данных с динамическим набором значений. Почти в каждом приложении нужен хотя бы один такой класс.

Даже в первых примерах составных классов, обсуждавшихся в этой главе (таких как класс Rectangle), содержалась совокупность компонентов (в данном случае — объектов класса Point), однако она не была динамической. Число объектов Point, связанных с объектом Rectangle, всегда одинаково и равно двум. Если бы два объекта Point были недоступны в клиенте, невозможно было бы создать объект Rectangle.

Часто в приложении необходим класс-контейнер или класс-набор, содержащий переменное число объектов. Обычно контейнерный объект пуст и не содержит компоненты. При выполнении приложения объекты-компоненты становятся доступными и добавляются в контейнер для временного хранения или для обработки.

Например, контейнер может представлять собой предъявляемый к оплате счет заказчика, а компоненты — подлежащие обработке операции с кредитной картой заказчика. Обработка может включать в себя вычисление общей суммы, налогов, вывод отчетов и выполнение других задач. Кроме того, можно проверить наличие в контейнере искóмых данных, удалить компоненты из контейнера и обеспечить повторное использование контейнера.

Большинство подобных задач можно выполнять с помощью представления данных в виде массива C++. Это очень простой и эффективный тип данных, но он имеет мало средств защиты для клиента, например, он не проверяет допустимости значений индексов. В нем нет таких операций высокого уровня как добавление компонента, поиск компонента и т. д. Эти операции нужно программировать в клиенте с помощью элементарных операций низкого уровня, например присваивание значения элементу массива, установка индекса на следующий элемент и проверка следующего элемента массива (является ли он допустимым).

Контейнерные классы проектируются так, что сами выполняют эти операции от имени клиента. Клиентская программа запрашивает у контейнера добавление компонента, поиск компонента, доступ к каждому компоненту в наборе, а контейнерный объект выполняет подобные операции, изолируя клиента от деталей низкого уровня. В результате обязанности переносятся с клиента на сервер (контейнерный класс).

В данном разделе приводится ряд примеров простых контейнеров. Показано, как их можно использовать в клиенте.

В примерах применяются компоненты класса `Sample`, содержащего всего один элемент данных `value` типа `double`. Объекты `Sample` генерируются внешним процессом. Это могут быть котировки акций, данные устройства мониторинга, значения температуры и давления и т. д. В примерах значения берутся из массива с заранее заданными элементами.

```
class Sample {                    // класс компонента
  double value;                   // указателей среди элементов данных нет
public:
  Sample (double x = 0)           // конструкторы: по умолчанию/преобразования
    { value = x; }
  void set (double x)             // модификатор
    { value = x; }
  double show () const            // селектор
    { return value; } } ;
```

Когда класс используется как компонент контейнера, он должен отвечать требованиям архитектуры. Наиболее распространенные требования — это способность поддерживать:

- Создание экземпляров по умолчанию
- Присваивание

Под требованием создания экземпляров по умолчанию понимается возможность создавать объекты типа компонента без каких-либо данных от клиента. Контейнер может использовать контейнерный класс для представления данных компонентов как массива фиксированного размера.

```
Sample data[100];                 // компонент данных контейнера
```

Еще одна популярная архитектура контейнерного класса — использование динамически распределяемого массива компонентов.

```
Sample *data;                    // элемент данных контейнера
data = new Sample[100];          // код в конструкторе контейнера
```

В любом случае сначала выделяется память для объектов-компонентов, а позднее они заполняются данными. Для поддержки этого требования класс компонента должен реализовывать используемый по умолчанию конструктор. В противном случае определение массива объектов-компонентов даст синтаксическую ошибку.

В конечном счете попытка создания составного объекта без конструктора по умолчанию класса компонента приводит к синтаксической ошибке. Ее можно устранить, включив в конструктор контейнера список инициализации компонентов. Этот метод работает, только когда число компонентов в контейнере известно заранее и не очень велико. Однако в списке инициализации компонентов нужно перечислять каждый компонент составного класса, явно указывая имя компонента.

Подобно предыдущим примерам, класс `Sample` может отвечать этому требованию, предлагая конструктор преобразования с заданным по умолчанию значением своего единственного параметра.

Требованием присваивания называют возможность присваивать в клиенте объекту компонентного типа новое состояние. Поскольку память для компонентов контейнера сначала распределяется, а данными заполняется позднее, эти компоненты должны поддерживать изменение состояния. Одним из популярных способов реализации такого требования является поддержка для компонента класса операции присваивания.

Этот метод хорошо использовать для простых классов вида `Sample`, поскольку в них не применяется динамическая память. Если же класс компонента содержит указатели и работает с динамически распределяемой памятью, может потребоваться перегруженная операция присваивания.

Еще один метод поддержки присваивания состоит в реализации для класса компонента функции-модификатора, которая будет изменять состояние объекта-компонента.

```
data[i].set(s);                  // код в методе контейнера
```

Класс `Sample` поддерживает это требование, предусматривая метод `set()` и допуская прямое присваивание без использования перегруженной операции присваивания.

Часто предполагается, что класс компонента должен отвечать еще двум требованиям, т. е. поддерживать:

- Создание копий экземпляров объектов
- Семантику общего порядка

Создание копий экземпляров — это способность создавать экземпляр объекта-компонента из другого объекта-компонента. Это свойство поддерживается с помощью реализации в классе компонента конструктора копирования и необходимо, если контейнерный класс должен возвращать копию одного из своих объектов-компонентов клиенту. Часто клиента устраивает не полноценная копия объектов-компонентов, а ссылка на них. Создание экземпляров-копий в алгоритмах контейнеров происходит редко. Его поддержка неизбежно поощряет передачу параметров-объектов по значению или возврат значений-объектов из функций со всеми негативными последствиями, поэтому лучше не включать в класс конструктор копирования и надеяться на лучшее. Вы столкнетесь с проблемами, особенно, если класс работает с динамически распределяемой памятью. Если не допускается передача объектов по значению или возврат значений-объектов, лучше применять частные конструкторы (см. главу 11).



Советуем Не торопитесь снабжать каждый класс конструктором копирования. Они усложняют программы и замедляют их работу, поощряют клиентов передавать параметры по значению и возвращать объекты по значению. Вместо этого сделайте конструктор копирования закрытым. Таким образом вы сможете предотвратить появление проблем.

Семантикой общего порядка называется способность клиента сравнивать объекты-компоненты между собой и со значениями встроенных типов. Для этого в классе компонента предусматриваются перегруженные операции сравнения. Данное свойство очень полезно, особенно когда в клиенте требуется реализовать механизмы сортировки и поиска.

```
История измерений:

  3  5  7  11  13  17  19  23
Среднее значение: 12.25
```

Рис. 12.7. *Результат выполнения программы из листинга 12.4*

В следующих примерах объекты класса Sample хранятся в контейнерном объекте класса History. Класс History хранит объект Sample в коротком массиве (для простоты в нем всего лишь восемь элементов). Это позволяет клиенту устанавливать значение Sample в заданном месте массива, выводить набор измерений и вычислять среднее для измеренных значений.

Листинг 12.4 показывает первую версию контейнерного класса. Результат программы представлен на рис. 12.7.

Листинг 12.4. Контейнерный класс с массивом компонентов фиксированного размера

```
#include <iostream>
using namespace std;

class Sample {                              // класс компонента
  double value;                             // значение для примера
public:
  Sample (double x = 0)                     // конструктор: по умолчанию и преобразования
    { value = x; }
  void set (double x)                       // метод-модификатор
    { value = x; }
  double get () const                       // метод-селектор
    { return value; } } ;

class History {                             // контейнерный класс
  enum { size = 8 };
  Sample data[size];                        // массив значений (фиксированного размера)
public:
  void set(double, int);                    // модификация значения
  void print () const;                      // вывод предыстории
  void average() const;                     // вывод среднего значения
 } ;

void History::set(double s, int i)
{ data[i].set(s); }                         // или просто: data[i] = s;

void History::print () const                // вывод предыстории
{ cout << "\n История измерений:" << endl << endl;
  for (int i = 0; i < size; i++)            // локальный индекс
    cout << " " << data[i].get(); }

void History::average () const
{ cout << "\n Среднее значение: ";          // вывод среднего значения
  double sum = 0;                           // локальное значение
  for (int i = 0; i < size; i++)            // локальный индекс
          sum += data[i].get();
  cout << sum/size << endl; }
```

```
int main()
{  double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;        // исходные данные
   History h;                                               // конструктор по умолчанию
   for (int i=0; i < 9; i++)                                // доступно 8 слотов
      h.set(a[i],i);                                        // установка предыстории
   h.print();                                               // вывод предыстории
   h.average();                                             // вычисление среднего значения
   return 0;
   }
```

Обратите внимание, что архитектура контейнерного класса `History` реализует алгоритм, требующий доступа к памяти. Это означает, что программа сохраняет некоторые значения в памяти для будущего использования в другом сегменте кода. В зависимости от расположения этих взаимодействующих сегментов программы разработчику приходится использовать те или иные виды их связывания.

Как правило, у разработчика класса есть выбор, позволяющий ему сделать значение или переменную доступной методу класса для хранения или получения значения:

- Глобальная переменная или общедоступный элемент данных
- Параметр метода
- Элементы данных класса
- Локальная переменная в методе

Глобальная переменная может использоваться, когда несколько классов должны совместно работать с информацией, но разработчик затрудняется определить, к какому из них относится подобное сообщение. Такой метод коммуникаций между классами представляет наивысшую степень связности, и применять его следует по возможности реже. Общедоступные элементы данных можно использовать, когда разработчик выбирает для хранения информации конкретный класс, но эта информация нужна и другим классам. Она будет доступна им в форме элемента данных `public`. Такая степень связи столь же высока, как и при использовании глобальной переменной, и не следует применять ее часто.

Когда несколько классов программы взаимодействуют через глобальные переменные или общедоступные элементы данных, это всегда следует рассматривать как повод для пересмотра архитектуры. Разработчикам необходимо проверить распределение обязанностей между классами. Коммуникации через глобальные переменные или общие элементы данных должны наводить на мысль, что, возможно, разработчик разделил шаги обработки, которые должны быть объединены. Если сделать все правильно, не исключено, что необходимость в таких "дальних" коммуникациях исчезнет.

Остановимся на трех других методах коммуникаций, поскольку именно между ними приходится ежедневно выбирать программисту, использующему C++.

Коммуникации через параметры метода следует применять в том случае, если значение или переменная будут совместно использоваться классом и его клиентом. Например, параметры метода `History::set()` совместно применяются клиентом `main()` и функцией-членом `set()` класса `History`.

Это наивысшая форма связи через данные, когда два разных класса используют одно общее значение. Согласованная интерпретация такого значения в двух разных классах требует кооперации и координации усилий между разработчиками классов. Когда над такими классами работает один программист, возможно, это происходит в разное время, поэтому ему потребуется помнить множество разных ограничений.

Про возможности такую форму связи следует свести к минимуму, ограничив значение или переменную одним классом и исключив подобные коммуникации между классами. Впрочем, это не всегда реально, поскольку объектно-ориентированная программа строится как набор взаимодействующих, а не полностью независимых классов. Следовательно, некоторые коммуникации между классами вполне законны и полезны. Тем не менее программист, работающий с C++, должен помнить о взаимодействии классов и стараться не использовать такую форму связности.

Если общую переменную нужно использовать в разных методах, принадлежащих одному классу, применяйте коммуникации через элементы данных класса. Например, класс Sample предусматривает для каждого объекта класса память для хранения элемента данных value. На этой стадии изучения C++ такое архитектурное решение должно вызывать у вас беспокойство, но нужно иметь в виду, что оно поддерживает коммуникации между двумя методами Sample — set() и get(). Какое бы значение ни устанавливала функция set() (например, при вызове set() в классе History), оно сохраняется с течением времени. Когда клиент класса Sample позднее вызывает функцию get() (например, в методах print() или average() в классе History), функция get() получает значение, сохраненное в этом конкретном объекте Sample его методом set().

Элемент данных data[] в классе History используется для коммуникаций между функциями-членами History. Какое бы значение ни устанавливала функция set() класса History, функции print() и average() будут применять именно его. К этому можно прийти другими способами. Например, определить массив date[] в main() как локальную переменную или глобальную переменную в файле и передавать его методам класса History как параметр.

Посмотрите на следующую версию класса History. Она выполняет все те же функции, что и версия из листинга 12.4, но не хранит в качестве своего компонента данных массив объектов Sample. Вместо этого класс History получает данные для работы от клиента main().

```
class History {
    enum { size = 8 };
public:
    void set(Sample[]; double, int) const;          // модификация значения
    void print (const Sample[]) const;              // вывод предыстории
    void average(const Sample[]) const;             // вывод среднего значения
  } ;

void History::set(Sample data[], double s, int i) const
{ data[i].set(s); }                                 // или просто: data[i] = s;

void History::print (const Sample data[]) const     // вывод предыстории
{ cout << "\n История измерений:" << endl << endl;
  for (int i = 0; i < size; i++)                    // локальный индекс
    cout << " " << data[i].get(); }

void History::average (const Sample data[]) const
{ cout << "\n Среднее значение: ";                 // вывод среднего значения
  double sum = 0;                                   // локальное значение
  for (int i = 0; i < size; i++)                    // локальный индекс
       sum += data[i].get();
  cout << sum/size << endl; }
```

Как бы ни была плоха данная архитектура, она синтаксически корректна и семантически надежна. Ее недостаток — чрезмерные коммуникации между классом History и клиентом. Клиенту приходится поддерживать информацию, которая в листинге 12.4 использовалась классом History.

```
int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;        // 9 значений
  Sample data[9];                    // кому должны принадлежать эти данные?
  History h;                         // конструктор по умолчанию
  for (int i=0; i < 9; i++)          // доступно 8 слотов
    h.set(data,a[i],i);              // установка предыстории
  h.print(data);                     // вывод предыстории
  h.average(data);                   // вычисление среднего значения
  return 0; }
```

Незначительные ошибки такого рода накапливаются, в результате ухудшается качество программ C++. Продумывая архитектуру, не забывайте о коммуникациях между классами.

Последний способ коммуникаций в программе C++ — взаимодействие через локальную переменную в методе. Его следует применять, когда функции-члену нужно сохранить значение для будущего использования в течение того же вызова. Например, функция average() в листинге 12.4 использует локальные переменные sum и i для отслеживания обработанных компонентов массива и их накапливаемого остатка на данный момент выполнения.

Подобно предыдущему примеру, такая конструкция может реализовываться по-разному. Рассмотрим следующую версию класса History, предусматривающую специализированные элементы данных для отслеживания компонентов и хранения остатка.

```
class History {                    // контейнерный класс
  enum { size = 8 };
  Sample data[size];               // массив значений (фиксированного размера)
  int i;                           // индекс для метода average()
  double sum;                      // остаток для метода average()
public:
  void set(double, int);           // модификация значения
  void print () const;             // вывод предыстории
  void average() const;            // вывод среднего значения
 } ;

void History::set(double s, int i)
{ data[i].set(s); }                // или просто: data[i] = s;

void History::print ()             // модифицирует i
{ cout << "\n История измерений:" << endl << endl;
  for (i = 0; i < size; i++)       // глобальный, а не локальный индекс
      cout << " " << data[i].get(); }

void History::average ()           // модифицирует sum
{ cout << "\n Среднее значение: ";          // вывод среднего значения
  sum = 0;                         // глобальное значение
  for (i = 0; i < size; i++)       // глобальный индекс
       sum += data[i].get();
  cout << sum/size << endl; }
```

В данной версии метод average() обращается к глобальным переменным (элементам данных) sum и i, а не к автоматическим переменным, распределяемым во время выполнения метода. Конечно, это отражается на производительности. При каждом вызове функции average() не нужно выделять память, поэтому данная версия работает быстрее. С другой стороны, память выделяется для каждого объекта History на время его существования, а не только для выполнения функции average(). Такую версию average() написать легче — можно использовать доступные переменные. Кроме того, их можно повторно применять в других функциях, например в print().

Одним из важных вопросов является качество ПО. В данном варианте все не так плохо, как в случае использования для взаимодействия глобальных переменных других функций. Функция `average()` применяет глобальные переменные (элементы данных) `sum` и `i` для взаимодействия с собой (на следующей итерации цикла), а не с другими функциями. Тем не менее такая конструкция свидетельствует о низком качестве программирования, и ее следует избегать. Один из примеров потенциальных осложнений — желание использовать глобальные элементы данных для других целей (подобно использованию индекса в функции `print()`, чтобы избежать описаний переменных), а это часто ведет к конфликтам. C++ поддерживает следующую теорию разработчика ПО: "Пусть каждая функция использует свои отдельные локальные переменные и применяет их так, как считает необходимым, без риска возникновения каких-либо конфликтов".

Нужно стремиться к тому, чтобы степень сопряжения отдельных фрагментов в программе была минимальной. Если этого можно добиться с помощью локальных переменных в одной компонентной функции, не следует "поднимать" такие переменные до уровня компонентных данных класса. Если нескольким компонентным функциям одного класса нужно обращаться к одним и тем же данным, реализуйте их как элементы данных класса и не передавайте как параметры. Когда функции-члену нужны данные, определенные в другом классе, передавайте их через параметры и не применяйте глобальную переменную или общедоступные элементы данных другого класса.



Советуем Для взаимодействия отдельных сегментов программы C++ используйте связность через локальные переменные в методе. Если для поддержки потока данных этого не достаточно, применяйте функции-члены класса. И только когда и их не хватает, передавайте информацию через параметры метода. В любом случае старайтесь не использовать глобальные переменные.

Применим эти принципы разработки ПО к архитектуре класса из листинга 12.4. Класс `History` — упрощенный контейнерный класс. Он не предусматривает никакой защиты клиента от переполнения контейнера или от ссылки на несуществующий элемент массива. Данная версия контейнера имеет только восемь слотов (ячеек) для хранения объектов `Sample`. Несмотря на это ограничение, клиент в `main()` помещает в контейнер девять значений. Конечно, компилятор такое поведение не волнует. Операционная система также выполняет программу без всяких проблем, хотя она некорректна (см. рис. 12.7). Это распространенная проблема для приложений, использующих контейнеры. Распределение обязанностей между клиентом и контейнерным классом может быть различным, но защита контейнера от переполнения должна быть реализована, что является обязанностью класса-контейнера, а не клиента.

Когда новое значение `Sample` помещается в контейнер, клиент в листинге 12.4 задает и само значение, и индекс, который будет использоваться для вставки. Между тем этот подход противоречит принципу разработки ПО, который состоит в переносе обязанностей в серверный класс (в данном случае — контейнер `History`). Возможно, для такого простого алгоритма это не важно (все вводимые значения поступают сразу, не прерывая операции объекта-контейнера), однако у клиента есть другие важные обязанности. Он не должен отслеживать, сколько места осталось в контейнере. Мониторингом состояния контейнера должен заниматься сам объект-контейнер.

Скорее всего, интерфейс между клиентом и `History::set()` прочно связаны. Клиент в дополнительном параметре вынужден передавать информацию об индексе. Согласно правилам взаимодействия классов, следующая по силе степень взаимодействия — коммуникации через элементы данных класса. Чтобы усовершенствовать имеющуюся архитектуру, нужно хранить информацию об индексе

следующего объекта Sample в классе History, а не в клиенте. Необходимо координировать работу программистов, если разделено то, что должно быть вместе.

Улучшенная архитектура контейнера представлена в листинге 12.5. Метод History::set() с двумя параметрами заменен на метод History::add() с единственным параметром — значением, добавляемым в конец контейнера. Контейнер содержит один дополнительный элемент данных — индекс idx, позволяющий отслеживать используемую контейнером память. Клиент не знает, полон контейнер или нет. Он просто передает методу add() добавляемое значение.

Так как клиент теперь не следит за использованием памяти в контейнере, отслеживание занятой и доступной памяти и контроль за переполнением являются обязанностями контейнера. Соответственно, контейнер знает о структуре своей памяти и ограничениях. В версии, представленной в листинге 12.4, где клиент должен решить, куда поместить следующее значение, не было необходимости инициализировать контейнерный объект. В данной версии, где контейнер сам решает, куда попадает следующее значение, он должен инициализироваться пустой областью, что обеспечит попадание в первый слот первого значения. Следовательно, класс History имеет используемый по умолчанию конструктор. В нем History устанавливает индекс idx в значение 0. В методе add() контейнерный класс проверяет, заполнен ли массив. Если есть свободное место, add() использует еще один свободный слот и увеличивает значение индекса idx для ссылки на следующий свободный слот. Если слоты для поступающих данных недоступны, метод add() ничего не делает и игнорирует запрос клиента.

Конечно, было бы хорошо сообщить клиенту, успешна ли попытка добавления значения Sample в History. Клиент смог бы инициировать некоторые меры восстановления или уведомить о ситуации пользователя программы. Однако программисту не стоит тратить на это время. Все массивы фиксированного размера следует применять лишь для быстрого макетирования, а после отладки алгоритма заменить их на динамические массивы (см. главу 6).

Результат программы из листинга 12.5 будет тем же, что и программы из листинга 12.4.

Листинг 12.5. Контейнерный класс с массивом фиксированного размера и контроль за переполнением

```
#include <iostream>
using namespace std;

class Sample {                          // класс компонента
  double value;                         // значение для примера
public:
  Sample (double x = 0)                 // конструктор: по умолчанию и преобразования
    { value = x; }
  void set (double x)                   // метод-модификатор
    { value = x; }
  double get () const                   // метод-селектор
    { return value; } } ;

class History {                         // контейнерный класс
  enum { size = 8 };
  Sample data[size];                    // массив значений (фиксированного размера)
  int idx;                              // индекс текущего значения
public:
  History() : idx(0) { }                // массив первоначально пуст
  void add(double);                     // добавление значения в конец
  void print () const;                  // вывод предыстории
  void average() const;                 // вывод среднего значения
 } ;
```

```
void History::add(double s)
{ if (idx < size)
     data[idx++].set(s); }                    // или просто: data[idx++] = s;

void History::print () const
{  cout << "\n История измерений:" << endl << endl;
   for (int i = 0; i < size; i++)             // локальный индекс
     cout << " " << data[i].get(); }

void History::average () const
{  cout << "\n Среднее значение: ";
   double sum = 0;                            // локальное значение
   for (int i = 0; i < size; i++)             // локальный индекс
     sum += data[i].get();
   cout << sum/size << endl; }

int main()
{  double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;        // исходные данные
   History h;                                 // конструктор по умолчанию
   for (int i=0; i < 9; i++)                  // доступно 8 слотов
     h.add(a[i]);                             // добавление значения к предыстории
   h.print();                                 // вывод предыстории
   h.average();                               // вычисление среднего значения
   return 0;
   }
```

Контейнерный класс History демонстрирует клиенту минимально возможную часть своей внутренней структуры и ограничений памяти.

Одно из важных ограничений контейнерного класса в листингах 12.4 и 12.5 состоит в том, что он должен быть заполнен до своей емкости, и лишь после этого клиент получает осмысленный доступ к компонентам в контейнере. Методы контейнера print() и average() итеративно перебирают массив до конца контейнера. Еще одно важное ограничение заключается в том, что с точки зрения клиента все операции с компонентами выполняются как одна операция. Между тем, нередко клиенту нужно обращаться к компонентам индивидуально, выполняя или пропуская операции для каждого компонента.

Первый недостаток можно устранить, добавив к контейнерному классу еще один элемент данных — count.

```
class History {                    // контейнерный класс
  enum { size = 8 };
  Sample data[size];               // массив значений (фиксированного размера)
  int count;                       // число действительных элементов
  int idx;                         // индекс текущего значения
public:
  History() : count(0), idx(0) { }          // массив первоначально пуст
  void add(double);                // добавление значения в конец
  ...} ;
```

В конструкторе (в списке инициализации) компонент данных count устанавливается в значение 0. Обновлять count следует при каждом добавлении нового компонента в контейнер.

```
void History::add(double s)
{ if (count < size                 // проверка доступного места
    data[count++]set(s); }         // следующая ячейка, обновление счетчика count
```

Если даже контейнер не полон, его компонентные функции могут использовать значение count для обработки корректного числа компонентов. Ниже функция average() применяет его для ограничения числа участвующих в вычислении компонентов.

```
void History::average () const
{ cout << "\n Среднее значение: ";
  double sum = 0;
  for (int i = 0; i < count; i++)
       sum += data[i].get();
  cout << sum/count << endl; }
```

Второй недостаток можно исключить, введя метод-итератор, позволяющий клиенту обращаться по любому поводу к каждому компоненту в контейнере. Такие итераторы реализуются разными способами.

Для перебора компонентов используется существующий элемент данных idx, который устанавливается в 0 в начале итерации и увеличивается на 1 на каждом шаге. Чтобы начать итерацию для клиента, добавим в контейнерный класс метод getFirst().

```
void getFirst()
  { idx = 0; }                    // устанавливается на начало набора данных
```

Чтобы клиент мог сделать следующий шаг итерации, добавим в контейнерный класс метод getNext():

```
void getNext()
  { ++idx; }                      // перемещение к следующему элементу
```

Чтобы клиент мог обращаться к текущему компоненту в контейнере, добавим метод getComponent():

```
Sample& getComponent()
  { return data[idx]; }          // получение ссылки, а не значения
```

Обратите внимание, что здесь возвращается не текущий объект, а ссылка на него. Таким образом, нужно позаботиться о том, чтобы класс компонента имел конструктор копирования. В этом случае можно не беспокоиться, что копирование компонента займет слишком много времени.

Чтобы остановить итерацию, нужен метод, возвращающий true, когда итерация может продолжаться, и false, когда в контейнере больше нет элементов для итерации.

```
bool atEnd()
  { return idx < count; }         // true, если остались еще элементы
```

Тогда цикл итерации в клиенте может выглядеть так:

```
for (h.getFirst(); h.atEnd(); h.getNext())          // перебор до конца
  cout << " " << h.getComponent().get();            // вывод компонентов
```

Часто разработчики контейнеров объединяют функции getNext() и atEnd() в одну функцию, увеличивающую индекс и возвращающую true, если больше элементов для итерации нет.

```
bool getNext()
{ return ++idx < count; }         // переход к следующему элементу в наборе
```

Листинг 12.6 показывает версию контейнерного класса с методами-итераторами. Здесь удален метод контейнера `print()`, а клиент отвечает за управление итерацией и доступ к элементам компонента. В результате некоторые обязанности переносятся с контейнера на клиента. Это не очень хорошо, но является естественным следствием добавления в контейнерный класс итератора.

Результат программы из листинга 12.6 будет тем же, что и результаты программ из листингов 12.4 и 12.5.

Листинг 12.6. Контейнерный класс с массивом фиксированного размера и итератором

```
#include <iostream>
using namespace std;

class Sample {                                  // класс компонента
  double value;                                 // значение для примера
public:
  Sample (double x = 0)                         // конструктор: по умолчанию и преобразования
    { value = x; }
  void set (double x)                           // метод-модификатор
    { value = x; }
  double get () const                           // метод-селектор
    { return value; } } ;

class History {                                 // контейнерный класс
  enum { size = 8 };
  Sample data[size];                            // массив значений (фиксированного размера)
  int count;                                    // число действительных элементов
  int idx;                                      // индекс текущего значения
public:
  History() : count(0), idx(0) { }              // массив первоначально пуст
  void add(double);                             // добавление значения в конец
  Sample& getComponent()                        // возвращает ссылку на Sample
  { return data[idx]; }                         // может быть целью сообщения
  void getFirst()
  { idx = 0; }                                  // установка на начало набора данных
  bool getNext()
  { return ++idx < count; }                     // переход к следующему элементу в наборе
  void average() const;                         // вывод среднего
} ;

void History::set(double s)
{ if (count < size)
    data[count++].set(s); }                     // или просто: data[i++] = s;

void History::average () const
{ cout << "\n Среднее значение: ";              // вывод среднего значения
  double sum = 0;
  for (int i = 0; i < count; i++)
          sum += data[i].get();
  cout << sum/count << endl; }

int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;          // исходные данные
  History h;                                  // конструктор по умолчанию
  for (int i=0; i < 9; i++)
    h.add (a[i]);                             // добавить историю
  cout << " /n Measurement history:" << end1 << end1;
  h.getFirst ();
```

```
do {
    cout << "" << h.getComponent().get();
  } while (h.getNext());
h.average();                                  // вычисление среднего значения
return 0;
}
```

Некоторые программисты предпочитают связывать методы-итераторы с отдельным итераторным классом и ассоциируют его с классом-контейнером. В данном примере это не делается, чтобы не усложнять программу. Устраним свойственное подобной архитектуре контейнера ограничение на число компонентов, которые он может содержать. В предыдущих версиях контейнера его емкость была фиксированной и задавалась во время создания контейнера. Если клиент пытался поместить в контейнер больше элементов, чем тот мог содержать, то, увы, контейнер с этим ничего не мог поделать.

На самом деле проблема легко решается. Контейнерный класс должен выделить новое пространство, скопировать в него дополнительные данные, освободить задействованную память и использовать новую область до тех пор, пока она не будет исчерпана. Хорошей стратегией для выделения новой памяти является удвоение ее объема (размера массива).

```
void History::add(double s)
{ if (count == size)
  { size = size * 2;                    // удвоение размера, если нет памяти
      Sample *p = new Sample[size];
    if (p == NULL)
      { cout << " Нет памяти\n"; exit(1); }          // проверка на успех
    for (int i=0; i < count; i++)
      p[i] = data[i];                  // копирование существующих элементов
    delete [ ] data;                   // удаление существующего массива
    data = p;                          // замена его на новый массив
    cout << " новый размер: " << size << endl; }       // отладка
  data[count++].set(s); }
                          // использование следующего доступного пространства
```

Чтобы алгоритм работал, элемент данных data должен обозначать динамически распределяемый массив объектов Sample, что требует изменений в конструкторе.

```
class History {                     // контейнерный класс: установка значения
    int size, count, idx;
    Sample *data;                   // динамическая память
  public:
    History() : size(3), count(0), idx(0)       // сделать массив пустым
    { data = new Sample[size];      // выделение новой памяти
      if  (data == NULL)
          { cout << " Нет памяти\n"; exit(1); } }
    . . . } ;                       // остальная часть класса History
```

```
новый размер: 6
новый размер: 12

История измерений:

  3  5  7  11  13  17  19  23  29
Среднее значение: 14.1111
```

Рис. 12.8. *Результат выполнения программы из листинга 12.7*

Данная версия контейнера показана в листинге 12.7. Для простоты примера в качестве начального размера контейнера задается очень маленькое значение (три компонента). Показана работа алгоритма. Результат программы продемонстрирован на рис. 12.8. Сначала выводятся отладочные сообщения об увеличении размера контейнера с 3 до 6 (когда в контейнер помещается четвертый элемент), а затем с 6 до 12 (когда в контейнере оказывается седьмое значение).

Листинг 12.7. Контейнерный класс с динамически распределяемой памятью

```
#include <iostream>                          // динамический контейнер переменного размера
using namespace std;

class Sample {                               // класс компонента
  double value;                              // значение для примера
public:
  Sample (double x = 0)                      // конструктор: по умолчанию и для преобразования
    { value = x; }
  void set (double x)                        // метод-модификатор
    { value = x; }
  double get () const                        // метод-селектор
    { return value; } } ;

class History {                              // контейнерный класс: установка значения
  int size, count, idx;
  Sample *data;
public:
  History() : size(3), count(0), idx(0)      // сделать массив пустым
  { data = new Sample[size];                 // выделение новой памяти
    if (data == NULL)
      { cout << " Нет памяти\n"; exit(1); } }
  void add(double);                          // добавление значения в конец
  Sample& getComponent()                     // возвращает ссылку на Sample
  { return data[idx]; }                      // может быть целью сообщения
  void getFirst()
  { idx = 0; }
  bool getNext()
  { return ++idx < count; }
  void average () const;                     // вывод значения
  ~History() { delete [ ] data; }            // возвращает динамическую память
} ;

void History::add(double s)
{ if (count == size)
  { size = size * 2;                         // удвоение размера, если нет памяти
      Sample *p = new Sample[size];
    if (p == NULL)
      { cout << " Нет памяти\n"; exit(1); }          // проверка на успех
    for (int i=0; i < count; i++)
      p[i] = data[i];                        // копирование существующих элементов
    delete [ ] data;                         // освобождение существующей памяти
    data = p;                                // замена на новый массив
    cout << " новые размеры: " << size << endl; }     // отладочный вывод
  data[count++].set(s); }                    // использование следующей доступной области

void History::average () const
{ cout << "\n Среднее значение: ";
  double sum = 0;
  for (int i = 0; i < count; i++)
      sum += data[i].get();
  cout << sum/count << endl; }

int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;      // исходные данные
  History h;
  for (int i=0; i < 9; i++)
    h.add(a[i]);                             // добавление значения к предыстории
```

```
cout << "\История измерений:" << endl; << endl;
h.getFirst();                                    // перенос обязанностей
do {
    cout << " " << h.getComponent().get();       // вывод каждого компонента
  } while (h.getNext());
h.average();
return 0;
}
```

Обратите внимание, что при уничтожении объекта-контейнера динамическое управление памятью требует для деструктора возврата динамической памяти использования.

Нетрудно придумать и более сложные конструкции: компоненты можно сортировать, искать в контейнере, удалять, вставлять, обновлять и сравнивать. Значительное число классов-контейнеров содержится в библиотеке Standard Template Library.

Вложенные классы

Вернемся к программе из листинга 12.5. Обратите внимание, что вместо добавления функций-итераторов в данной версии программы клиент не использует объекты Sample.

```
int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;      // исходные данные
  History h;                                             // конструктор по умолчанию
  for (int i=0; i < 9; i++)                              // доступно 8 слотов
    h.add(a[i]);                                         // установка предыстории
  h.print();                                             // вывод предыстории
  h.average();                                           // вычисление среднего
  return 0; }
```

Вместо этого к классу Sample имеет доступ объект History. C++ позволяет программисту определить серверный класс внутри клиентского класса. В результате имя вложенного класса будет невидимо вне составного, агрегатного класса.

```
class History {
  class Sample {                    // невидим вне области действия клиента
    double value;                   // закрытые данные: здесь они могут
                                    // быть общедоступными
  public:
    Sample (double x = 0)
      { value = x; }
    void set (double x)
      { value = x; }
    double show () const { return value; }
      } ;                           // конец определения вложенного класса
    int size, count, idx;
    Sample *data;
public:
  History() : size(3), count(0), idx(0)       // сделать массив пустым
  { data = new Sample[size];                  // выделение новой памяти
    if (data == NULL)
      { cout << " Нет памяти\n"; exit(1); } }
    . . . } ;                       // остальная часть класса History
```

Определения вложенных классов могут включаться в зарытые или общедоступные части клиентского класса. В любом случае вложенный класс скрыт от остальной программы. Имя класса известно только в области действия (фигурных скобок) композитного класса, где определен вложенный класс.

Вложенные классы не могут использоваться для объявления переменных, имеющих тип этих классов, вне области действия составного класса. Определения класса скрываются таким же образом, как и элементы данных. Следовательно, если имя нужно использовать вне того класса, где оно определено, применяйте операцию области действия.

```
int main()
{ double a[] = {3, 5, 7, 11, 13, 17, 19, 23, 29 } ;     // исходные данные
  History h;                        // конструктор по умолчанию
  for (int i=0; i < 9; i++)         // доступно 8 слотов
    h.add (a[i]);                   // установка предыстории
    h.print();                      // вывод предыстории
    h.average();                    // вычисление среднего
  Sample s = 5;                     // не допускается для вложенного класса
  History::Sample s = 5;            // операция области действия решает проблему
  return 0; }
```

Чтобы последний оператор был допустимым, класс `Sample` нужно определять в разделе `public` клиентского класса `History`. Тогда клиент класса `History` сможет использовать имя `Sample`, уточненное именем агрегатного класса.

C++ позволяет комбинировать в одном операторе определения класса и его экземпляры. Между тем это нельзя считать хорошей практикой программирования, поскольку такие экземпляры часто являются глобальными.

```
class Sample {                          // глобальный в файле
    double value;
  public:
    Sample (double x = 0)
       { value = x; }
    void set (double x)
       { value = x; }
    double show () const { return value; }
       } s1,s2;                         // глобальные объекты класса Sample
```

В то же время это вполне подходит для вложенных классов, так как компонентные данные обычно глобальны в области действия класса.

```
class History {
  class Sample {                    // невидим вне области действия клиента
    double value;
  public:
    Sample (double x = 0)
       { value = x; }
    void set (double x)
       { value = x; }
    double show () const
       { return value; }
    } *data;                        // конец определения вложенного класса
  int size, count, idx;
public:
  History() : size(3), count(0), idx(0)      // сделать массив пустым
  { data = new Sample[size];        // выделение новой памяти
```

```
    if (data == NULL)
       { cout << " Нет памяти\n"; exit(1); } }
. . . } ;                          // остальная часть класса History
```

Не следует злоупотреблять данным средством, ибо в результате может получиться запутанная программа.

Если в других классах нужно использовать эти классы компонентов как их серверные классы, то не следует определять класс компонента как вложенный. Когда другие классы не нуждаются в классах компонентов, вложенные классы не дают никаких преимуществ, однако их применение позволяет избежать конфликтов имен, если в нескольких частях программы для совершенно разных целей будет использоваться одно и то же имя класса. Тогда имя вложенного класса не будет "загрязнять" пространство имен программы.

Например, в других классах для определения иных результатов измерений других типов с другого датчика и даже числа значений может использоваться имя Sample. Для разрешения такого конфликта надо ввести два имени класса, например Sample1 и Sample2. Однако удобнее использовать одно имя и сделать класс вложенным.

Такие имена классов, как Node, очень популярны для компонентов связанного списка. Если некоторый класс Node полезен для другой связанной структуры (например, стека или двоичного дерева), то его следует объявлять в глобальном пространстве имен. Однако нередко разные связанные структуры содержат разнообразные элементы информации, и один класс Node не может их обслуживать.

В данной ситуации надо определить класс Node как локальный в каждом из контейнерных классов. При этом устраняются потенциальные конфликты имен. Устранение глобальных имен ведет к тому, что программистам, разрабатывающим разные контейнерные классы, не нужно координировать свои действия.

```
struct Node {          // хороший кандидат на вложенный класс
  char* value;         // указатель на содержимое информации (например, слово)
  Node* next; } ;      // указатель на следующий узел
```

"Дружественные" классы

Разработчик класса должен предоставить доступ к своему классу, чтобы удовлетворить потребности клиентов, не открывая класс и не создавая ненужные зависимости.

Размещение элементов данных в секции public класса используется довольно редко. При этом предоставляется чрезмерно широкий доступ без явного указания того, кто его использует. Если возникает проблема при отладке, не ясно, какие именно клиенты должны подлежать проверке. Когда меняется представление данных, также не вполне очевидно, на каких клиентов это повлияет. Использование закрытых данных и функций возможно только через общедоступные функции-члены.

Доступ к отличным от public компонентам класса через функции-члены может усложнить исходный код клиента. Как было показано в главе 11, C++ предлагает способ для расширения доступа к частным разделам класса. "Дружественная" функция класса имеет те же полномочия доступа, что и компонентная функция класса.

Это нарушает инкапсуляцию и должно использоваться по возможности реже. С другой стороны, список "дружественных" функций и классов — часть явного определения класса. Он доступен для проверки и при необходимости может использоваться для идентификации клиентов, на которых влияют изменения. Так как "дружественная" функция обращается к закрытым объектам класса, практически бесполезно применять ее вне контекста класса.

"Дружественными" могут быть не только "автономные" глобальные функции, но и компоненты другого класса. В этом случае функция-член одного класса будет "дружественной" функцией другого.

Можно определить все функции одного класса как функции friend другого класса. Тогда функции-члены этого класса смогут обращаться к закрытым компонентам другого класса без помощи функций доступа.

Чаще "дружественными" функциями серверного класса объявляются только некоторые функции-члены клиента.

Когда класс используется как сервер только одним клиентским классом, архитектуру клиента и сервера можно упростить, сделав клиентский класс "дружественным" для серверного класса. Тогда каждая функция-член класса-клиента (например, History) сможет обращаться к отличным от public компонентам серверного класса (например, Sample). Данный синтаксис предусматривает применение ключевого слова friend, предшествующего имени класса (в любой части серверного класса).

```
class Sample {
              friend History;          // описание friend
    double value;
  public:
       Sample (double x = 0)
           { value = x; }
       void set (double x)
           { value = x; }
       double show () const
           { return value; }
       } ;
```

Если класс History еще не определен — это синтаксическая ошибка. Между тем, класс History не определяется перед классом Sample, так как в нем используется имя Sample.

```
class History {
    int size, count, idx;
    Sample *data;                              // циклическая зависимость
public:
  History() : size(3), count(0), idx(0)        // пустой массив
  { data = new Sample[size];                   // выделение новой памяти
    if (data == NULL)
       { cout << " Нет памяти\n"; exit(1); } }
    . . . } ;                                  // остальная часть класса History
```

Это типичный пример циклической зависимости в программе. Класс History использует имя Sample, следовательно, определение Sample перед определением класса History. С другой стороны, класс Sample применяет имя History, и, следовательно, необходимо определять History перед определением класса Sample.

Существуют два разных способа, позволяющих сообщить компилятору, что означает History в определении Sample. Один из них — применить упреждающее описание, при котором указывается, что имя является именем класса. Здесь имя History определяется как имя класса.

```
class History {                // класс объявлен в другом месте

class Sample {
       friend History;         // объявление friend
    double value;
```

```
  public:
    Sample (double x = 0)
       { value = x; }
    void set (double x)
       { value = x; }
    double show () const
       { return value; }
    };
```

Другой способ решить эту проблему — указать непосредственно в объявлении friend, что History является классом.

```
class Sample {
       friend class History;              // объявление friend
    double value;
  public:
    Sample (double x = 0)
       { value = x; }
    void set (double x)
       { value = x; }
    double show () const
       { return value; }
    };
```

Конечно, хорошо было бы иметь только один способ, а еще лучше, если бы компоновщик мог сам разрешать эти перекрестные ссылки.

Теперь функции-члены History могут непосредственно обращаться к отличным от public данным.

```
void History::print () const
{ for (int i = 0; i < count; i++)          // вывод только допустимых элементов
//      cout << " " << data[i].show();
      cout << " " << data[i].value;        // не нужно использовать методы
    cout << endl; }
```

Более того, классу Sample теперь не требуются функции-члены для предоставления доступа, его функция friend в них не нуждается.

```
class Sample {
       friend class History;       // объявление friend
    double value;
  public:
    Sample (double x = 0)
       { value = x; }              // другие функции не нужны
       };
```

В технике разработки ПО и методологии программирования с подозрением относятся к компонентам friend, поскольку они намеренно нарушают принцип сокрытия информации. Кроме того, они усложняют архитектуру программы и затрудняют обучение программированию.

Вместо применения элементов friend класс Sample можно сделать вложенным в класс History и объявить общедоступные поля. Поскольку никакой другой класс, кроме History, к этим полям не обращается, принцип сокрытия информации соблюдается.

```
class History {
  intsize, count, idx;
struct Sample {                        // невидим вне клиента
```

```
scope
    double value;                              // элемент данных public
  Sample (double x = 0)
    { value = x; } } *data;                    // динамические данные History
public:
  History() : size(3), count(0), idx(0)        // сделать массив пустым
  { data = new Sample[size];                   // выделить новую память
    if (data == NULL)
       { cout << " Нет памяти\n"; exit(1); } }
  . . . } ;                                    // остальная часть класса History
```

Реализация функций-членов History при этом будет такой же, как в предыдущей версии. Они имеют полный доступ к отличным отpublic компонентам Sample.

Используйте элементы friend осторожно и всегда рассматривайте другие альтернативы.

Итоги

В данной главе рассматривалось использование классов C++ как компонентов композиции классов. Классы следует определять как соотносящиеся друг с другом взаимодействующие компоненты, а не считать их автономными сегментами программного кода.

Агрегирование — одна из наиболее популярных форм связи между классами. Использование объектов классов как элементов данных ставит ряд технических и концептуальных вопросов: как определить компоненты класса, как их инициализировать соответствующим состоянием для использования в клиенте (компонентном классе).

Рассматривались также другие способы связи объектов — применение указателей и ссылок. Это еще более сложная техника связей между объектами. Подробное обсуждение данных методов программирования слишком далеко увело бы вас от изучения синтаксиса C++. Однако, как бы не сложилась ваша карьера профессионального программиста, вам придется включать одни объекты в другие и связывать их с помощью указателей и ссылок.

Кроме того, отмечался специальный случай агрегирования классов — контейнерные классы, содержащие набор компонентов-объектов. Это также очень эффективное соотношение между классами, которое можно реализовать несколькими способами. Данные способы предусматривают создание контейнерных классов или использование библиотечных контейнеров (либо сочетание того и другого).

Надеемся, вам будет приятно и интересно с ними работать.

Глава 13

Подобные классы и их интерпретация

Темы данной главы

- ✔ Интерпретация подобных классов
- ✔ Синтаксис наследования в C++
- ✔ Доступ к сервису базовых и производных классов
- ✔ Доступ к базовым компонентам объекта производного класса
- ✔ Правила области действия и разрешения имен при наследовании
- ✔ Конструкторы и деструкторы производных классов
- ✔ Итоги

В данной главе, как и в предыдущей, читатели более подробно познакомятся с синтаксисом C++ — ключевыми словами, списками инициализации и др.

В первой части книги рассказывалось о вычислительных аспектах C++. Она была посвящена таким традиционным темам программирования, как типы данных, идентификаторы, ключевые слова, выражения, операторы, условия, циклы и другие управляющие структуры. Применение этих инструментов позволяет достичь стоящих при написании программы целей и получить результаты, отвечающие требованиям вычислений.

Кроме того, говорилось о методах агрегирования — включения компонентов данных в массивы, структуры и другие определяемые программистом типы, объединении операторов и управляющих конструкций в функции. В C++ это сложнее, чем в других языках, особенно когда дело касается областей действия, передачи параметров, возврата значений, указателей и ссылок. Читатели узнали также о возможностях и опасностях динамического управления памятью в C++. Эти инструментальные средства предполагают разбиение программы на взаимодействующие части. В то же время они более удобны для программиста. Вычислительных целей программы можно достичь с помощью множества альтернативных архитектурных решений, но качество программы при этом будет различаться.

Навыки корректного комбинирования элементов программы C++ и разделения тех компонентов, которые не следует объединять вместе, являются необходимой предпосылкой для написания легко сопровождаемых и модифицируемых программ C++.

Вторая часть книги была посвящена применению знаний, полученных в ее первой части, для написания классов C++. В ней рассказывалось о синтаксисе классов, области их действия, элементах данных и функциях, доступе к данным и функциям, сообщениях (их синтаксисе и смысле), инициализации объектов, о статических данных и функциях. Читатели узнали об операторных функциях, за счет применения которых программа на C++ становится эффективной, однако они усложняют архитектуру классов. Кроме того, вы познакомились с "дружественными" функциями, научились распознавать в конструкции класса опасные элементы и избегать их негативного влияния на программу. Написание программы с использованием классов усложняет C++, но оно того стоит.

Навыки объединения (включения в один класс) имеющих отношение друг к другу данных и функций являются необходимой предпосылкой написания объектно-ориентированных программ. Основное различие между традиционными и объектно-ориентированными программами состоит в том, что традиционные программы C++ строятся с помощью взаимодействующих глобальных функций, связывающих шаги каждой операции. Объектно-ориентированная программа состоит из взаимодействующих классов, объединяющих данные и операции.

Первые две части книги — это введение к объектно-ориентированному программированию. Во всех примерах использовался только один класс, поскольку основное внимание уделялось архитектуре классов, а не связям между ними. В третьей части книги вы приступили к изучению построения программ C++ как набора взаимодействующих классов. Это требует реализации связей между классами программы. В главе 12 было показано, как можно использовать один объект (сервер) в качестве компонента другого объекта (клиента). Функции-члены объекта-компонента предоставляют свой сервис функциям-членам составного класса. Это наиболее распространенная простая связь между объектами.

Один объект может ссылаться на другой, являющийся элементом данных следующего объекта. Клиентский объект получает доступ к функциям-членам серверного объекта, передавая сообщения его элементу данных (указателю). Кроме того, один объект может использоваться как ссылочный элемент данных объекта-клиента. Синтаксически такой вариант аналогичен простой композиции классов, но фактически это совершенно разные связи между объектами.

При простой композиции класса серверный объект (компонент) является элементом данных объекта-клиента (составного объекта). При такой связи клиентский объект имеет эксклюзивные права на объект-компонент. Когда серверный объект представляет ссылочный элемент данных объекта-клиента, он может совместно использоваться несколькими клиентами, и несколько объектов могут ссылаться на один серверный объект. Изменения в серверном объекте влияют на состояние клиентского объекта. Вряд ли имеет смысл обсуждать, какая связь "лучше" — эксклюзивная композиция классов или совместное использование компонентов. Между тем во многих практических ситуациях одна связь действительно "лучше" другой, так как точнее представляет выраженную в программе C++ взаимосвязь между реальными объектами. Поэтому важно выбирать те связи, которые лучше моделируют реальную ситуацию.

Кроме того, рассматривалась такая популярная связь между объектами, как контейнер, когда один объект включает в себя набор объектов другого класса — компонентов контейнера (а не один единственный объект). Такая взаимосвязь часто встречается в программах C++, и нужно хорошо освоиться с организацией объектов в приложении и с выбором подходящих связей между ними.

В данной главе продолжается исследование взаимодействия между фрагментами программы C++. Здесь читатели познакомятся с наследованием как механизмом, представляющим связь между классами в приложении. На этом этапе разница между связью объектов и классов может казаться вам незначительной, но к концу главы вы ее почувствуете.

Наследование очень часто применяется в программах C++. Оно представляет мощный механизм для повторного использования фрагментов программ C++, для разделения труда программистов и использования модульности в приложениях. Для корректного применения наследования следует изучить его синтаксис, методы создания экземпляров производных объектов, технику доступа к компонентам, правила вызова функций, разрешения имен и многое другое. Важно также знать, когда использовать наследование. Программисты, применяющие C++, иногда злоупотребляют наследованием. Они создают дополнительные взаимосвязи и зависимости, которые затрудняют понимание программы.

Будьте внимательны.

Интерпретация подобных классов

Программы моделируют различные реальные объекты через их данные (состояние объекта) и операции (его поведение). Это азбука объектно-ориентированного проектирования ПО, но каждый разработчик должен сам решить, что именно следует включать в каждый класс. Моделирование реальных ситуаций должно отражать "общие свойства" объектов, например инвентарных записей, счетчиков событий или банковских счетов.

Все "общие свойства" с точки зрения наблюдателя, и C++ предусматривает разнообразные механизмы для объектов.

Первый механизм, предлагаемый C++ для воплощения общих свойств реальных объектов в классе, это сама конструкция класса. Она используется для показа общности объектов. Вы верите, что эти объекты можно охарактеризовать одним и тем же набором атрибутов и шаблонами поведения. Объекты различаются значениями атрибутов: точки разных углов фигуры имеют разные координаты, разные инвентарные единицы — различные названия, на каждом банковском счете — свой собственный баланс, и у них разные владельцы. Общность в том, что каждый прямоугольник имеет угловые точки, каждая инвентарная единица — название, каждому счету соответствует баланс и владелец. Если для одного банковского счета требуется задавать проценты, а для другого — нет, то обычно они не рассматриваются как объекты одного класса.

Часто ситуация бывает не ясна. Например, каждый болт в инвентарной описи имеет индивидуальные характеристики и может отличаться в приложении от других болтов. Чтобы описать каждый болт, придется создать для него отдельный класс, присвоить ему индивидуальный набор элементов данных и функций, задать уникальные имена. Такие имена могут отражать уникальный характер каждого болта в приложении, например `RustyBolt`, `UglyBolt` и `BoltFoundInPothole`. Не исключено, что это осложняет ситуацию и имеет смысл только в случае, если отдельные болты не обладают общими свойствами и ведут себя по-разному.

Между тем у болтов в инвентарной описи много общего, так что можно представить каждый болт, используя для элементов данных одни и те же имена. Таким образом, не нужно будет представлять каждый болт как объект другого класса. Возникновение проблемы можно предотвратить, используя всего лишь один класс, например `Bolt`, и представляя каждый болт в приложении как объект данного класса с такими атрибутами, как дата покупки, имя поставщика и шаг резьбы. Аналогично, можно представить все гайки в инвентарной описи как объекты одного класса `Nut`, если для описания каждого такого объекта достаточно одного и того же набора атрибутов (цвет, материал, размер и т. д.).

Если класс `Bolt`, `Nut` и другие инвентарные элементы используют одни и те же имена элементов данных, нужно применить только один класс `InventoryItem`, представляющий эти разные объекты. Если все болты с точки зрения приложения одинаковы, можно представить их как один объект и указать в числе атрибутов класса количество болтов. Так как все болты одинаковы, разница в шаге резьбы не столь важна. Если же она имеет значение, такая конструкция не подойдет.

Если приложение интересует лишь общая стоимость болтов, гаек и других инвентарных единиц, можно представить инвентарную опись как объект типа Asset с атрибутами, соответствующими целям приложения.

Часто между классами существует общность. Группы их объектов могут быть в чем-то похожи, но различаться набором атрибутов и операций.

Например, для мелких болтов может задаваться вес партии в 100 штук, а для больших болтов — вес одной штуки. Иногда важно также значение максимального усилия, которое должно применяться к крупному болту.

Точно так же для сотрудников, получающих почасовую оплату, в качестве элементов данных может указываться зарплата в час с числом наработанных в неделю часов. У сотрудников с месячным окладом многие атрибуты будут такими же (фамилия, адрес, дата приема на работу и т. д.), но вместо почасовой оплаты задается годовая.

Некоторые группы объектов могут различаться набором операций или осуществлять дополнительные операции. Например, сберегательный счет предусматривает выплату процентов, а расчетный — плату за операции. Если все эти характеристики собрать в один класс, такой вариант будет отвечать требованиям клиента, но окажется ненадежным. Клиент может некорректно использовать объект, предположив наличие у него свойств, присущих другому объекту, например, он может попытаться выплатить проценты по расчетному счету или взять плату за операцию по сберегательному счету.

Тем не менее слияние всех атрибутов и операций в один класс — вполне жизнеспособный метод абстракции. Именно клиент должен убедиться, что каждый объект используется согласно его характеристикам.

Слияние свойств подклассов в один класс

Рассмотрим такой пример.

Возьмем упрощенный класс Account с элементом данных balance и функциями-членами withdraw() и deposit(). Для расчетного счета операция снятия денег должна предусматривать плату (например, 20 центов). Для сберегательного счета ежедневно начисляются проценты (6% годовых). Уровень платы за операцию и проценты годовых представлены в классе Account как элементы данных. Для простоты примера здесь не обсуждаются методы спецификации и изменения числовых литералов и другие бесчисленные практические детали, например фамилия владельца, адрес, возраст, номер социального страхования, плата за превышение кредита и другие неприятные и приятные детали банковского бизнеса.

В листинге 13.1 показана программа, реализующая свойства обоих счетов в комбинированном классе Account. Клиент определяет объекты Account и выполняет соответствующие операции. Такой вид клиента типичен для "дообъектно-ориентированного" программирования.

Листинг 13.1. Пример комбинирования различных свойств в одном классе Account

```
#include <iostream>
using namespace std;

class Account {

  double balance;                          // для всех видов счетов
  double rate;                             // только для сберегательных
  double fee;                              // только для расчетных

public:

  Account(double initBalance = 0)          // только для расчетных счетов
  { balance = initBalance; fee = 0.2; }    // использовать плату за операцию, а не %
```

```
    Account(double initBalance, double initRate)        // для сберегательного счета
    { balance = initBalance; rate = initRate; }         // не плата, а %

    double getBal()
    { return balance; }                                 // для обоих счетов

    void withdraw(double amount)                        // общая для обоих счетов
    { if (balance > amount)
         balance -= amount; }

    void deposit(double amount)                         // для обоих счетов
    { balance += amount; }

    void payInterest ()                                 // только для сберегательных счетов
    { balance += balance * rate / 365 / 100; }

    void applyFee()
    { balance -= fee; }                                 // только для расчетных счетов
    } ;

int main()
{
    Account a1(1000), a2(1000,6.0);                     // a1: расчетный счет, a2: сберегательный счет
    cout << "Начальный баланс: " << a1.getBal()
         << " " << a2.getBal() << endl;
    a1.withdraw(100); a2.deposit(100);                  // нет проблем
    a2.payInterest(); a1.applyFee();                    // нет ошибок
    cout << "Конечный баланс: " << a1.getBal()
         << " " << a2.getBal() << endl;
    return 0;
    }
```

Сегодня никто не верит в человеческую непогрешимость. Если что-то где-то может быть написано, то когда-нибудь так и случится. Например, пятая строка клиента может выглядеть так:

```
a1.payInterest();  a2.applyFee();                       // неудачная попытка...
```

Конечно, невозможно предотвратить все ошибки программирования (вот почему необходимо тестирование), но следует хотя бы постараться это сделать. Вы можете добиться того, чтобы сообщения об ошибках выводились заранее и не нужно было анализировать результат. Данная архитектура нуждается в улучшении.

Обратите внимание, что в клиенте при создании счета даются явные комментарии о его типе, но ничто не мешает здесь программисту выразить свои идеи в операторах, а не в комментариях. Для этого серверный класс (`Account`) должен поддерживать потребности клиента, позволяя ему явно различать виды объектов `Account`.

Перенос ответственности за целостность программы на сервер

Чтобы избежать опасности некорректного использования серверных объектов клиентом, можно добавить в серверный класс дополнительный атрибут — поле тега (признака), описывающее вид счета, к которому относится данный конкретный объект. Это означает, что в классе вводятся подклассы.

При создании объекта поле тега можно установить таким образом, чтобы указать подкласс объекта при его инициализации. При использовании объекта (например, `payInterest()` или `applyFee()`) это поле проверяется, чтобы убедиться в допустимости операции для данного вида объекта.

Например, при создании объекта Account можно установить поле тега в 0, если объект будет использоваться как расчетный счет. Если планируется работать с объектом как со сберегательным счетом, можно установить поле тега в 1. Это означает, что конструктор должен каким-то образом определять, какой именно создается объект Account.

В данном примере предположим, что конструкторы для двух разных видов объектов должны иметь разное число параметров. Кроме того, использование числового значения поля тега — не очень хорошая практика программирования. Разработчик знает, что означают 0 или 1 в этом поле, а другие программисты могут запутаться. Как сообщить о поле разработчику? Для этого в C++ используются перечисления. Сделаем поле Kind типа перечисления локальным для класса Account. Поскольку тип Kind не предполагается использовать вне класса Account, удобно сделать Kind вложенным в него. Данное имя не будет "загрязнять" глобальное пространство имен, и его смогут использовать другие программисты, работающие над проектом.

```
class Account {
  enum Kind { CHECKING, SAVINGS } ;                    // константы вида счета
  double balance;
  double rate, fee;
  Kind tag;                                            // поле тега для вида объекта
public:
  Account(double initBalance = 0)                      // расчетный счет
  { balance = initBalance; fee = 0.2;
    tag = CHECKING; }
  Account (double initBalance, double initRate)  // сберегательный счет
  { balance = initBalance; rate = initRate;
    tag = SAVINGS; }
  . . . } ;                                        // остальная часть класса Account
```

Если бы вам улыбнулась фортуна, тип Kind мог бы быть доступным также и для клиента, который мог бы явно указывать вид создаваемого счета. Это означает, что конструктор должен включать в себя параметр вида счета.

Давайте усложним пример, предположив, что начальные проценты по вкладам для всех сберегательных счетов одинаковы, и, следовательно, их не нужно задавать в клиентском коде. Таким образом, классу Account требуется всего один конструктор. Клиенту нужно задавать вид объекта счета, и тип Kind следует сделать глобальным ("загрязнив" пространство имен и увеличив степень взаимодействия между разработчиками). Вот как выглядит новый класс Account:

```
enum Kind { CHECKING; SAVINGS } ;                    // константы вида счета

class Account {
  double balance;
  double rate, fee;
  Kind tag;
public:                                              // поле тега для вида объекта
  Account(double initBalance, Kind kind)             // только один конструктор
  { balance = initBalance; tag = kind;               // задание поля тега
    if (tag == CHECKING)
       fee = 0.2;
    else if (tag == SAVINGS)
       rate = 6.0; }                                 // сберегательный счет
  . . . } ;                                        // остальная часть класса Account
```

Обратите внимание, что мы стараемся не использовать одну и ту же область памяти для процентов по вкладу, если это объект сберегательного счета, и платы за операцию, если это объект расчетного счета. Если приложение работает

с большим числом объектов Account в памяти и память является критическим ресурсом, то такой вариант заслуживает внимания. В противном случае в программном коде появятся дополнительные зависимости. Подобного альтернативного использования памяти следует избегать.

Теперь клиент явно использует перечисления для указания вида создаваемого объекта Account. Заметим, что комментарии теперь стали лишними. Они лишь повторяют то, что уже выражено в самом программном коде, поэтому идеи разработчика достаточно эффективно передаются сопровождающему приложение программисту.

```
Account a1(1000,CHECKING);          // a1 - расчетный счет
Account a2(1000,SAVINGS);           // a2 - сберегательный счет
```

"Загрязнения" пространства имен перечислением типа Kind можно избежать, даже если клиенту нужно использовать значения данного типа (как в приведенных выше примерах). Один из способов добиться этого — сделать тип локальным в классе Account:

```
class Account {
  double balance;
  double rate, fee;
  Kind tag;
public:                                          // поле тега для вида объекта
  enum Kind { CHECKING; SAVINGS } ;              // константы вида счета
  Account(double initBalance, Kind kind)         // только один конструктор
  { balance = initBalance; tag = kind;           // задание поля тега
    if (tag == CHECKING)
       fee = 0.2;                                // расчетный счет
    else if (tag == SAVINGS)
       rate = 6.0; }                             // сберегательный счет
  . . . } ;                                     // остальная часть класса Account
```

Теперь клиенту при работе с литеральными значениями перечисления в аргументах конструктора придется использовать операцию области действия.

```
Account a1(1000,Account::Kind::CHECKING);       // a1 - расчетный счет
Account a2(1000, Account::Kind::SAVINGS);       // a2 - сберегательный счет
```

Чтобы эта конструкция работала, тип Kind нельзя определять как закрытую часть класса Account. Обратите внимание, что при использовании данного типа внутри класса Account (для элемента данных tag) не обязательно следовать определению типа. Хотя компиляторы C++ являются однопроходными, внутри определений классов они делают два прохода.

Это относится только к новым компиляторам. Некоторые старые компиляторы будут сообщать, что тип Kind в определении поля tag не определен. Для таких компиляторов определение типа Kind должно предшествовать определению поля tag. Чтобы в клиенте оно было видимым, его нужно включить в общедоступную часть определения класса (public). Для согласования этих противоречивых требований следует добавить в определение класса дополнительные секции public и private.

```
class Account {
  double balance;
  double rate, fee;
public:
  enum Kind { CHECKING; SAVINGS } ;             // константы вида счета
private:
  Kind tag;                                     // поле тега для вида объекта
```

```
public:
  Account(double initBalance, Kind kind)    // только один конструктор
  { balance = initBalance; tag = kind;      // задание поля тега
    if (tag == CHECKING)
       fee = 0.2;
    else if (tag == SAVINGS)
       rate = 6.0; }                        // сберегательный счет
  . . . } ;                                 // остальная часть класса Account
```

Если поле тега правильно инициализируется в конструкторе, разработчик класса Account может защитить клиента от несогласованностей. Чтобы программист, занимающийся клиентской частью, ошибочно не взял плату за операцию после вызова withdraw() для сберегательного счета, серверный класс Account проверяет характер объекта и применяет плату за операцию только к расчетному счету.

```
void withdraw(double amount)          // общее для обоих счетов
{ if (balance > amount)
    {  balance -= amount;
       if  (tag == CHECKING)          // только для расчетных счетов
           balance -= fee; } }
```

Как можно видеть, функциональность applyFee() теперь реализуется функцией-членом withdraw(), так что клиенту не нужно помнить, какой именно вид объекта следует вызывать. Остается надеяться, что читатели понимают, как действуют принципы сокрытия информации и переноса обязанностей на серверы.

Метод payInterest() проверяет, является ли объект-получатель сообщения сберегательным счетом. Если это так, то начисляются проценты за день. Если счет является накопительным, то выводится ошибка этапа выполнения, уведомляющая тестировщика, что программист сделал ошибку, вызвав функцию для неверного объекта. Операция прерывается.

Обратите внимание на терминологию. Именно создатель класса Account выполняет работу от имени клиента. В "дообъектно-ориентированном программировании" клиент вынужден защищать себя собственными силами (или обеспечивать отсутствие ошибок). Во времена объектно-ориентированного программирования обязанности переносятся с клиента на серверный класс. Это очень распространенный архитектурный подход. Надо научиться его использовать.

```
Начальные балансы: 1000  1000
Расчетный счет: недопустимая операция
Итоговые балансы: 899.8  1100.18
```

Рис. 13.1. *Результат программы из листинга 13.2*

Листинг 13.2 показывает реализацию класса Account, применяющего данную технику для проверки допустимости действий клиента. Обратите внимание, что тип Kind определен вне класса Account. Результат программы представлен на рис. 13.1.

Листинг 13.2. Пример проверки корректности операций клиента на этапе выполнения

```
#include <iostream>
using namespace std;

enum Kind { CHECKING, SAVINGS } ;              // константы для вида счета

class Account {
  double balance;
  double rate, fee;
  Kind tag;                                    // поле тега для вида объекта
public:
  Account(double initBalance, Kind kind)
  { balance = initBalance; tag = kind;         // установка поля тега
```

```
        if (tag == CHECKING)
           fee = 0.2;                          // для проверки счета
        else if (tag == SAVINGS)
           rate = 6.0; }                       // для сберегательного счета

    double getBal()
    { return balance; }                        // общая для обоих счетов

    void withdraw(double amount)               // общая для обоих счетов
    { if (balance > amount)
        { ( balance -= amount;
        if (tag == CHECKING)                   // только для проверки счетов
           balance -= fee; } }

    void deposit(double amount)
    { balance += amount; }

    void payInterest()                         // только для сберегательных счетов
    { if (tag == SAVINGS)
         balance += balance * rate / 365 / 100;
      else if (tag == CHECKING)
         cout << "Расчетный счет: недопустимая операция\n"; }
    } ;

int main()
{ Account a1(1000,CHECKING);                   // a1 - расчетный счет
  Account a2(1000,SAVINGS);                    // a2 - сберегательный счет
  cout << "Начальные балансы: " << a1.getBal()
       << " " << a2.getBal() << end1;
  a1.withdraw (100); a2.deposit (100);         // нет проблем
  a1.payInterest(); a2.payInterest();          // неплохо?
  cout << "Итоговые балансы: " << a1.getBal()
       << " " << a2.getBal () << end1;
  return 0;
  }
```

Так как тип Kind теперь глобальный, клиент может задавать вид счета с помощью одних идентификаторов CHECKING и SAVINGS в вызовах конструктора.

```
Account a1(1000,CHECKING);          // a1 - расчетный счет
Account a2(1000,SAVINGS);           // a2 - сберегательный счет
```

Конечно, это проще того, с чем приходилось работать ранее, когда тип Kind был локальным (Account::Kind::CHECKING и Account::Kind::SAVINGS).

Такой код проще писать, однако предыдущая версия показывает программисту, сопровождающему приложение, что литералы перечисления принадлежат классу Account и никакому другому классу. Хотя данная версия проще в написании, разработчик должен координировать использование глобального имени Kind с другими программистами, которым оно может потребоваться для других целей. Как уже говорилось в главе 1, в современном подходе к программированию предпочтительнее "многословная" запись, требующая больше операторов, а не более "компактный" вариант, если последний требует больше координации между программистами при разработке программы и больше усилий, чтобы разобраться в ней. Не пишите раздутые программы, но сравните код с требованиями легкого понимания программы.

Благодаря объединению данных и операций различных подтипов в одном классе каждый метод класса осуществляет контроль за своими операциями. Система не будет аварийно завершать работу, и есть возможность корректно закончить ее

в случае ошибки. Между тем, в сервере необходим дополнительный анализ типов. Каждый метод контролирует законность операций независимо от других, в соответствии со значением тега объекта. Для крупной системы с большим числом различных видов объектов такая зависимость от вида объекта делает код сервера слишком объемным.

Кроме того, класс Account содержит много лишней информации об интерпретации объектов разных подтипов (сберегательный и расчетный счета). Объем информации, с которой придется иметь дело разработчику и сопровождающему приложение программисту, слишком велик. Если потребуется добавить еще один вид (или подтип) объекта, следует расширить методы существующего класса. Поскольку это повлияет на другие части программного кода, не имеющие отношения к изменениям, проведите обширное регрессионное тестирование.

Основная проблема данного подхода в том, что ошибки программирования клиента будут проявляться на этапе выполнения, а не компиляции. Кому-то придется изучить все эти сообщения и контролировать клиентскую часть. Хорошо было бы сделать так, чтобы некорректное использование разных видов объектов приводило к синтаксическим ошибкам, а не к ошибкам на этапе выполнения.

Отдельные классы для каждого серверного объекта

Хороший способ решения данной проблемы — создание отдельных классов, чтобы каждый класс реализовывал специализированный класс, а не просто свойства всех подклассов объектов. В нашем примере это означает создание классов CheckingAccount и SavingsAccount.

Каждый из этих классов придется проектировать сначала. CheckingAccount содержит все необходимое для работы с расчетным счетом, без каких-либо попыток включить туда средства, связанные со сберегательным счетом.

```
class CheckingAccount {
  double balance;
  double fee;                                    // нет процентов
public:
  CheckingAccount(double initBalance)
  { balance = initBalance; fee = 0.2; }          // расчетный счет
  double getBal ()                               // общий для обоих счетов
  { return balance; }                            // общая для обоих счетов
  void withdraw(double amount)
  { if (balance > amount)
       balance = balance - amount - fee; }       // безусловная плата
  void deposit(double amount)
  { balance += amount; }
} ;
```

Аналогично, класс SavingsAccount содержит все необходимое для поддержки операций с накопительными счетами. В нем реализованы все нужные функции и не обращается внимание на потребности клиентов с расчетными счетами.

```
class SavingsAccount {
  double balance;
  double rate;                                   // нет платы за операцию
public:
  SavingsAccount(double initBalance)
  { balance = initBalance; rate = 6.0; }         // сберегательный счет
  double getBal()
  { return balance; }                            // общая для счетов
```

```
  void withdraw(double amount)
  { if  (balance > amount)          // тот же интерфейс, разный код
        balance -= amount; }
  void deposit(double amount)       // общее для счетов
  { balance += amount; }
  void payInterest()                // только для сберегательных счетов
  { balance += balance * rate / 365 / 100; }
} ;
```

Листинг 13.3 показывает исходный код данной программы, реализующий такой подход. Обратите внимание на отсутствие перечисления для типа Kind. Теперь нет необходимости ни в локальном, ни в глобальном аргументе. Хотя каждый вид счета использует при инициализации одно и то же число параметров, клиенту не нужно применять тип перечисления для указания вида создаваемого счета. Программа явно определяет объекты счетов a1 и a2 как объекты класса CheckingAccount или SavingsAccount. Следовательно, каждое определение объекта вызывает соответствующий конструктор CheckingAccount или SavingsAccount. Результат программы показан на рис. 13.2.

```
Начальные балансы: 1000  1000
Итоговые балансы: 899.8  1100.18
```

Рис. 13.2. *Результат из листинга 13.3*

Листинг 13.3. Пример отдельных классов для разных подтипов объектов

```
#include <iostream>
using namespace std;

class CheckingAccount {
  double balance;
  double fee;                                  // нет процентов

public:
  CheckingAccount(double initBalance)
  { balance = initBalance; fee = 0.2; }        // расчетный счет

  double getBal ()
  { return balance ;}                          // общий для обоих счетов

  void withdraw(double amount)
  { if (balance > amount)
        balance = balance - amount - fee; }    // безусловная плата

  void deposit(double amount)
  { balance += amount; }
} ;

class SavingsAccount {
  double balance;
  double rate;                                 // нет платы за операцию

public:
  SavingsAccount(double initBalance)
  { balance = initBalance; rate = 6.0; }       // сберегательный счет

  double getBal()
  { return balance; }                          // общая для счетов

  void withdraw(double amount)                 // тот же интерфейс, разный код
  { if (balance > amount)
        balance -= amount; }

  void deposit(double amount)                  // общее для счетов
  { balance += amount; }
```

```
  void payInterest()                          // только для сберегательных счетов
  { balance += balance * rate / 365 / 100; }
} ;

int main()
{
  CheckingAccount a1(1000);                   // a1 - расчетный счет
  SavingsAccount a2(1000);                    // a2 - сберегательный счет
  cout << "Начальные балансы: " << a1.getBal()
       << " " << a2.getBal () << end1;        // нет проблем
  a1.withdraw(100); a2.deposit(100);          // теперь это синтаксическая ошибка!
  // a1.payInterest();
  a2.payInterest();                           // это нормально
  cout << "Итоговые балансы: " << a1.getBal()
       << " " << a2. getBal () << end1;
  return 0;
  }
```

С помощью такой конструкции решается проблема ошибок клиента. Вместо ошибки этапа выполнения генерируется ошибка компиляции.

```
a1.payInterest();          // синтаксическая ошибка: метод не найден
```

Единственная проблема состоит в том, что идеи разработчика не очень хорошо передаются сопровождающему приложение программисту. Здесь вы видите два класса, у которых много общего: элемент данных баланса, операции снятия со счета, размещения вклада, доступ к данным. Однако сама архитектура программы не обозначает общность классов. Разработчик классов знает, что у них общие свойства, но в программе это никак не обозначается.

Классы имеют общие имена, но этого недостаточно для большой программы. В листинге 13.3 оба класса помещаются на одной странице в исходном файле, но в реальной жизни они могут разделяться. Если один класс изменяется, нет никакой гарантии, что будет изменен и другой. При увеличении в программе числа объектов разных видов общие свойства этих классов не идентифицируются. Знания разработчика не выражаются в исходном коде.

Применение наследования C++ для связывания родственных классов

Еще одним решением данной проблемы является наследование. Программист может создать класс, содержащий общие для всех подтипов свойства. В терминах объектно-ориентированного анализа и проектирования он представляет обобщение состояния и поведения данных подклассов. Тогда можно повторно использовать эти общие свойства в других специализированных классах. Каждый специализированный класс добавляет свои конкретные свойства к обобщенному классу.

Например, можно обобщить понятие накопительного и расчетного счета, введя понятие счета. Вместо соединения всех свойств и сохранения счетов в классе `Account` лучше включить в него только свойства, общие для обоих видов счетов. Подобные свойства — элемент данных `balance`, методы `getBal()`, `withdraw()` и `deposit()`.

```
class Account {                          // общие свойства базового класса
protected:
  double balance;
public:
  Account(double initBalance = 0)
  { balance = initBalance; }
```

```
  double getBal()
  { return balance; }                    // общая для обоих счетов
  void withdraw(double amount)           // общая для обоих счетов
  { if  (balance > amount)
        ( balance -= amount; }
  void deposit(double amount)
  { balance += amount; }
} ;
```

Единственная разница между классами C++, которые встречались ранее, и этим классом в том, что ключевое слово `private` заменено на ключевое слово `protected`. Это ключевое слово предотвращает доступ к компонентам класса извне подобно ключевому слову `private`, однако есть важное отличие: `protected` разрешает доступ наследникам данного класса.

В терминологии C++ класс, обобщающий свойства других классов и объединяющий их общие характеристики, называется *базовым* классом. Он используется как базовый класс для дальнейшего наследования. Специализированные классы, добавляющие новые свойства к общим свойствам, заданным в базовом классе, называются *производными* классами. В C++ термин "*производный*" означает "наследующий". В Java используется другое понятие — *расширение* (extension).

Кроме того, базовые классы называют суперклассами или *родительскими* классами, а производный класс — *дочерним*. В контексте, когда базовый класс определяет тип данных, производный тип называется *подтипом*.

Производные классы добавляют и иногда заменяют свойства обобщенного базового класса. Дополнительные данные и методы в производном классе отражают связь между классами.

Например, классы `CheckingAccount` и `SavingsAccount` конструируются как отдельные специализации обобщенного класса `Account`. Они добавляют свойства, относящиеся к взиманию платы за операцию и выплаты процентов по вкладу, которых обобщенный класс `Account` не имеет.

Производный класс `SavingsAccount` добавляет к базовому классу `Account` элемент данных `rate` и функцию-член `payInterest()`. Он использует элемент данных `balance` и функции-члены `getBal()`, `withdraw()`, `deposit()` базового класса, не заменяя ни на одно из этих свойств. Следующий фрагмент показывает, что нужно сделать для определения производного класса. Здесь не повторяются все свойства, наследуемые из базового класса. Описываются только те свойства, которые добавляются к свойствам базового класса или заменяют их на собственную версию. (Синтаксис наследования описан в следующем разделе.)

```
class SavingsAccount : public Account {      // производный класс
  double rate;
public:
  SavingsAccount(double initBalance)
  { balance = initBalance; rate = 6.0; }     // сберегательный счет
  void payInterest()                         // не для расчетных счетов
  { balance += balance * rate /365 / 100; } } ;
```

Производный класс `CheckingAccount` добавляет к классу `Account` элемент данных `fee`. Он использует элемент данных `balance` и функции-члены `getBal()` и `deposit()`, заменяет функцию-член базового класса `withdraw()` на собственную функцию `withdraw()`, которая, в отличие от `withdraw()` базового класса, взимает плату за операцию по счету.

```
class CheckingAccount : public Account {     // производный класс
  double fee;
```

```
public:
  CheckingAccount(double initBalance)
  { balance = initBalance; fee = 0.2 }          // расчетный счет
  void withdraw(double amount)
  { if (balance > amount)
       balance = balance - amount - fee; }     // не для сберегательного счета
};
```

Таким образом, на этапе разработки применение наследования становится инструментом для проектирования программы с учетом повторного использования ее компонентов. В каждом производном классе можно определить общие свойства класса Account,— в базовом классе. В результате архитектура программы становится более компактной (не нужно повторять общие свойства), а производительность разработчика повышается.

В данных примерах концепции счетов, служащих, инвентарных единиц представляли скорее абстрактные, чем реальные объекты, которые должны моделироваться в приложении. Имейте в виду, что у вас есть расчетные и сберегательные счета, болты и гайки, служащие на окладе и с почасовой оплатой.

Среди реальных объектов часто можно встретить "естественные" связи суперкласс/подкласс, которые отражаются в связях между классами. Например, каждый автомобиль — это транспортное средство, и каждый "Запорожец" — автомобиль. Такую связь можно выразить с помощью наследования.

Наследование может быть прямым или *косвенным*. Транспортное средство — *прямой* суперкласс или базовый класс автомобиля. Автомобиль — прямой суперкласс или базовый класс "Запорожца".

Вполне естественно, что класс (например, автомобиль) является производным классом одного класса (транспортное средство) и базовым другого ("Запорожец").

Кроме того, наследование можно использовать для дальнейшего развития программы. Если требуется реализовать более специализированные операции, в производном классе достаточно определить только то, что для этого необходимо. Все остальное предоставляет базовый класс.

Как и в случае любого распределения обязанностей между классами, наследование может использоваться как инструмент для разделения труда при разработке ПО. Один монолитный класс создает один программист, а базовый и производный классы — разные.

За счет применения общности классов можно писать меньше исходного кода, а увеличение модульности программы способствует разделению труда. В то же время нельзя с уверенностью сказать, что наследование всегда делает программу более компактной. Если базовый класс невелик, и программа содержит всего несколько подтипов, то размер исходного кода немного уменьшится. Если же базовый класс велик, подтипы отличаются разнообразием, а каждый подтип добавляет только несколько новых свойств, программа действительно будет значительно меньше, поскольку не придется повторять в каждом подклассе код базового класса.

На время написания производных классов CheckingAccount и SavingsAccount код базового класса Account замораживается. Это мощная парадигма управления проектами. Если в будущем класс Account изменится, распространение изменений на все производные классы произойдет автоматически.

Еще одной популярной областью применения наследования является связывание методов на этапе выполнения. Это называют также привязкой этапа выполнения, динамическим связыванием или полиморфизмом с виртуальными функциями. Многие считают, что объектно-ориентированное программирование заключается в использовании наследования и полиморфизма. Это не так.

Полиморфизм — особый случай объектно-ориентированного программирования, когда программа обрабатывает набор родственных объектов, выполняющих аналогичные, но не идентичные операции над различными видами объектов.

Данные виды объектов настолько похожи, что их можно получать как производные из общего базового класса (например, овал, прямоугольник, треугольник как производные от фигуры). Операции также похожи, поэтому в каждом классе для ее осуществления можно использовать одно имя (например, `draw()`).

Полиморфизм позволяет обрабатывать список объектов, отправлять одинаковое сообщение для каждого объекта независимо от того, к какому классу он принадлежит. В зависимости от класса, к которому относится тот или иной объект, вызывается та или иная функция, хотя формально вызов выглядит как вызов функции базового класса (виртуальная функция).

Синтаксис наследования в C++

Основной прием в использовании наследования в C++ — двоеточие, за которым следует имя производного класса. Оно обозначает место имени базового класса и описание режимов наследования — общедоступного (public), закрытого (private) или защищенного (protected).

```
Итоговые балансы
   объект Account:                1200
   объект расчетного счета:       1099.8
   объект сберегательного счета:  900.148
```

Рис. 13.3. *Результат программы из листинга 13.4*

Листинг 13.4. показывает программу из листинга 13.3, реализованную с применением наследования. Клиент представляет собой расширение клиента из листинга 13.3. Поэтому и результат программы (см. рис. 13.3) является расширением результата программы из листинга 13.3.

Листинг 13.4. Пример иерархии наследования для классов `Account`

```
#include <iostream>
using namespace std;

class Account {                                  // базовый класс иерархии
protected:
   double balance;

public:
   Account(double initBalance = 0)
   { balance = initBalance; }

   double getBal()
   { return balance; }                           // общая для обоих счетов

   void withdraw (double amount)
   { if (balance > amount)                       // общая для обоих счетов
        balance -= amount; }

   void deposit(double amount)
   { balance += amount; }
} ;

class CheckingAccount : public Account {         // первый производный класс
   double fee;

public:
   CheckingAccount(double initBalance)
   { balance = initBalance; fee = 0.2; }         // расчетный счет

   void withdraw(double amount)
   { if (balance > amount)
        balance = balance - amount - fee; }      // безусловная плата
} ;
```

```
class SavingsAccount: public Account {          // второй производный класс
  double rate;

public:
  SavingsAccount(double initBalance)
  { balance = initBalance; rate = 6.0; }        // сберегательный счет

  void payInterest()                            // не для расчетных счетов
  { balance += balance * rate / 365 / 100; }
} ;

int main()
{
  Account a(1000);                              // объект базового класса
  CheckingAccount a1(1000);                     // объект производного класса
  SavingsAccount a2(1000);                      // объект производного класса
  a1.withraw(100);                              // метод производного класса
  a2.deposit(100);                              // метод базового класса
  a1.deposit(200);                              // метод базового класса
  a2.withdraw(200);                             // метод базового класса
  a2.payInterest();                             // метод производного класса
  a.deposit(300);                               // метод базового класса
  a.withdraw(100);                              // метод базового класса
  // a.payInterest();                           // синтаксическая ошибка
  // a1.payInterest();                          // синтаксическая ошибка
  cout << "Итоговые балансы\n объект Account: "
       << a.getBal() << endl;
  cout << " объект расчетного счета: " << a1.getBal() << endl;
  cout << " объект сберегательного счета: " << a2.getBal() << endl;
  return 0;
  }
```

Различные режимы создания производного класса из базового класса

Для обозначения наследования можно использовать те же три ключевых слова, что и для предоставления прав на элементы данных класса: public, protected и private. Именно они (с предшествующим двоеточием) указывают, что между классами существует связь наследования.

Поскольку ключевые слова те же, многие программисты считают, что и смысл их при наследовании такой же, как при управлении доступом к компонентам класса. Например, в следующем фрагменте ключевое слово public используется дважды.

```
class CheckingAccount : public Account {      // Account - базовый класс
  double fee;          // в производный класс добавлен элемент данных
public:                // начало общедоступного сегмента данных
  . . . } ;            // остальная часть производного класса CheckingAccount
```

Не следует считать, что в обоих случаях употребление public имеет один и тот же смысл. Общее только само ключевое слово public и двоеточие. В случае управления доступом двоеточие находится справа от ключевого слова. Это значит, что к последующим элементам данных можно обращаться из любого места программы. В режиме наследования двоеточие находится слева от ключевого слова. Смысл ключевого слова заключается в том, что права доступа к наследуемым компонентам будут те же, что и в базовом классе, т. е. закрытый компонент в базовом классе остается закрытым в производном и т. д.

Права доступа к элементам данных класса и режим создания производного класса задаются одним ключевым словом. Однако имейте виду, что все остальное разное.

Двоеточие и ключевое слово для режима наследования синтаксически связывают базовый и производный классы. Независимо от того, где в исходном коде размещаются определения класса, у программиста, проверяющего его, есть неоднозначный визуальный признак. Он означает:

- Наличие другого класса, используемого как базовый для данного класса
- Имя базового класса

Если использовать диаграмму UML (Unified Modeling Language), то связи между классами обозначаются связями между значками классов с пустыми треугольными стрелками, указывающими вершинами на базовый класс. Если у базового класса более одного производного класса, то каждый производный класс может иметь индивидуальную связь с базовым классом или общую связь (с одной стрелкой). Альтернативные способы описания связей между классом Account и двумя производными классами показаны на рис. 13.4.

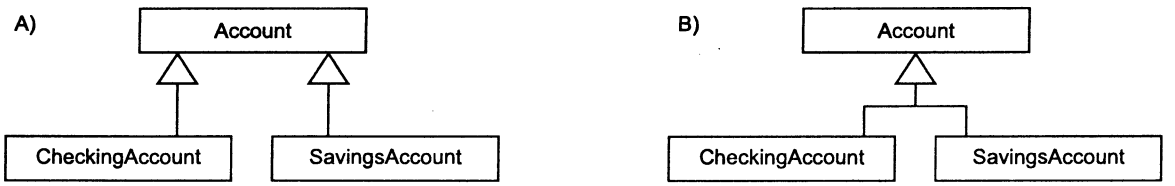


Рис. 13.4. *Связи между классами в иерархии Account*

Это пример использования наследования как способа представления связанных понятий приложения. Расчетный счет "является видом" счета (Account). Каждый расчетный счет — это счет, но не каждый счет — расчетный. Таково обобщенное замечание относительно связей наследования. Каждый автомобиль — транспортное средство, но не каждое транспортное средство — автомобиль. Прямоугольник есть вид многоугольника, но не каждый многоугольник — прямоугольник.

Отношение "является видом" ("is a") концептуально связывает классы и допускает применение наследования. Это отличается от агрегирования, когда объекты просто связываются отношением принадлежности. Например, прямоугольник имеет точки, а объект History содержит объекты Sample. Было бы некорректно говорить, что объект History является объектом Sample. У этих двух объектов совершенно разные данные и поведение. При наследовании данные и поведение классов также различны, но имеют общее подмножество, которое определено в базовом классе. Класс Account содержит элемент данных balance и метод deposit(). Благодаря наследованию класс CheckingAccount также имеет элемент данных balance и метод deposit(), хотя в определении класса эти компоненты не перечисляются.

Наследование представляет собой связь между классами. Класс Account определяет элемент данных balance, а класс CheckingAccount этого не делает. Так как класс CheckingAccount наследует свойства от класса Account, объекты CheckingAccount являются объектами Account и имеют все свойства Account, а также все свойства, указанные в определении класса CheckingAccount.

Таким образом, наследование не экономит память. Все данные Account присутствуют в каждом объекте CheckingAccount. Наследование помогает создавать компактные классы, если они становятся чрезмерно большими, и показывает логическую взаимосвязь между ними. Например, в листинге 13.4 показано, что классы CheckingAccount и SavingsAccount связаны. Оба они являются наследниками класса Account. В листинге 13.3 такую логическую взаимосвязь показать было

невозможно. В нем определения классов размещались вместе, но было показано наличие общих элементов данных и функций-членов. Программисту, читающему исходный код, приходилось додумываться до этого самому.

Каждый класс C++ можно использовать как базовый для создания производных классов. Иерархия наследования транзитивна. Например, из класса CheckingAccount можно получить класс TradingAccount. Объект TradingAccount будет иметь все свойства объекта CheckingAccount. Так как все объекты CheckingAccount имеют общие свойства объекта Account, объект TradingAccount включает также все свойства объекта Account.

С этой точки зрения термины "суперкласс" и "подкласс", часто применяемые для обозначения базового и производного класса, не очень точны. Они показывают, что базовый класс (суперкласс) в чем-то превосходит производный класс (подкласс), а это не так.

Возможности базовых классов не теряются ниже по иерархии, в производных классах. Объекты CheckingAccount могут делать все то же, что и объекты Account. Ниже по иерархии увеличиваются лишь ограничения членства. Класс CheckingAccount более ограничен, чем Account. В мире меньше расчетных счетов, чем счетов вообще. Аналогично, в мире меньше объектов TradingAccount, поскольку каждый объект TradingAccount является объектом CheckingAccount.

Рассматривая иерархию классов, можно видеть, что в каждом подклассе число экземпляров объектов уменьшается, но объектам данного подкласса становится доступно больше средств. С математической точки зрения число экземпляров во множестве может иметь важное значение, а с точки зрения программирования в расчет принимается предлагаемый объектом сервис. Суперклассы предлагают меньше сервисов, чем подклассы. Вот почему эти термины неточно отражают суть проблемы. Предпочтение отдается терминам "базовый класс" и "производный класс".

Наследование повышает модульность программного кода и способствует повторному использованию компонентов. Группу хорошо сконструированных классов общего назначения можно организовать в библиотеку. Интерфейс таких библиотечных классов следует опубликовать, а реализацию — инкапсулировать. Библиотечные классы могут специализироваться путем создания новых производных классов. В этих классах к элементам данных и функциям базового класса добавляются новые данные. Подобный метод широко используется для создания графических пользовательских интерфейсов. Классы приложения наследуют свойства из библиотечных классов — окон, диалоговых блоков, графических командных кнопок. Программисты приложения применяют эти свойства, реализованные в библиотечных классах, добавляют специфические свойства, которые определяют, как именно должна вести себя в приложении конкретная кнопка, диалоговый блок или окно.

В процессе такой специализации вносить изменения в базовые библиотечные классы не требуется. Следовательно, нет необходимости в их редактировании и перекомпиляции.

Как показывает листинг 13.4, каждый производный класс должен явно указывать свой базовый класс. Кроме того, в нем задаются дополнительные данные и функции.

```
class SavingsAccount : public Account {      // синтаксис производного класса
  double rate;                          // дополнительное средство
public:
  . . . } ;                             // остальная часть класса SavingsAccount
```

Между тем клиент ничего не должен знать о наследовании. Если клиент реализуется в отдельном файле, то в нем должен быть известен только производный класс, но не базовый. Базовый класс должен быть известен в тех файлах, где содержится его спецификация и где он реализуется. Это также подтверждает, что

наследование не является механизмом более качественного обслуживания клиента. Оно представляет собой механизм проектирования серверных классов (SavingsAccount и CheckingAccount). Как именно проектируются эти классы (с помощью наследования или с самого начала) для клиента не имеет значения.

Определение и использование объектов базовых и производных классов

Если клиенту требуется объект, можно определять и использовать объекты базового и производного класса. Если клиент находится в отдельном файле, то в него нужно включить заголовочные файлы каждого класса (в листинге 13.4 — для базового класса Account и двух производных классов SavingsAccount и CheckingAccount.

Какой метод вызывается в ответ на сообщение? Метод определяется в соответствии с объявленным типом целевого объекта. Компилятор задает целевой объект и ищет класс, к которому данный целевой объект принадлежит. В листинге 13.4 показаны все типичные ситуации в клиенте, которые нужно уметь распознавать.

```
Account a(1000);                                    // объект базового класса
CheckingAccount a1(1000);                           // объект производного класса
SavingsAccount a2(1000);                            // объект производного класса
a1.withdraw(100);                                   // метод производного класса
a2.deposit(100);                                    // метод производного класса
a1.deposit(200);                                    // метод базового класса
a2.withdraw(200);                                   // метод базового класса
a2.payInterest();                                   // метод базового класса
a.deposit(300);                                     // метод производного класса
a.withdraw(100);                                    // метод базового класса
// a.payInterest();                                 // синтаксическая ошибка
// a1.payInterest();                                // синтаксическая ошибка
cout << " Итоговые балансы\n объект Account: "
     << a.getBal() << endl;
cout << " объект CheckingAccount: " << a1.getBal() << endl;
cout << " объект SavingsAccount: " << a2.getBal() << endl;
```

Если целевой объект является объектом базового класса, компилятор генерирует вызов функции-члена, принадлежащей этому базовому классу.

```
a.deposit(300);                    // метод базового класса
```

Данное правило действует, даже если метод определен в производном классе и для объектов производного класса выполняется по-другому. Например, метод withdraw() по-другому задан для производного класса CheckingAccount. Тем не менее, когда надо получить объект базового класса, вызывается именно его метод withdraw().

```
a.withdraw(100);                   // метод базового класса
```

Обычно объекты базового класса ведут себя в клиенте так же, как при отсутствии производного класса. Они не могут отвечать на сообщения, определяемые в производных классах в дополнение к свойствам, унаследованным от базового класса. Например, попытка запросить у объекта Account выполнения задачи, присвоенной производному классу SavingsAccount, компилятором отвергается.

```
a.payInterest();                   // синтаксическая ошибка
```

Хотя классы Account и SavingsAccount связаны друг с другом через наследование, этого недостаточно для того, чтобы объект Account отвечал на сообщения

производного класса. Метод payInterest() в определении базового класса Account отсутствует, и вызов функции дает синтаксическую ошибку.

Другая ситуация возникает, когда получателем сообщения является объект производного класса. Нужно различать три случая:

1. Метод унаследован из базового класса и не переопределяется в производном классе.
2. Метод отсутствует в базовом классе и добавлен в производном классе.
3. Метод имеется в базовом классе и переопределен в производном классе.

Когда клиент вызывает унаследованный метод, у компилятора возникает проблема. Подобно обработке других сообщений, он находит тип получателя сообщения (вспоминает, что оно отправлено объекту производного класса) и ищет в спецификации производного класса имя функции-члена.

```
a1.deposit(200);                    // базовый метод класса
```

Очевидно, функции-члена там нет, поскольку унаследованные методы (в данном случае deposit()) описываются только в базовом, а не в производном классе. Описать унаследованный метод синтаксически приемлемо и в производном классе, но в таком случае это был бы уже не унаследованный, а переопределенный метод.

Когда метод не удается найти в классе целевого объекта, компилятору следует предупредить программиста об отсутствии вызванного метода. Однако перед этим компилятор проверяет, следует ли за спецификацией имени класса двоеточие. Если метод найден в базовом классе, он корректно приходит к заключению, что это производный класс, находит имя базового класса и ищет в нем определение. В противном случае компилятор проверяет, имеет ли этот класс базовый класс и повторяет процедуру, пока не произойдет одно из двух событий: в цепочке наследования будет найден класс без базового класса или в спецификации очередного класса обнаружится искомая функция. В последнем случае компилятор проверяет число и типы аргументов, сравнивает их с сигнатурой функции и генерирует для вызова функции объектный код.

Обратите внимание, что применение наследования нарушает первый принцип объектно-ориентированного программирования: связывание данных и операций в определении класса в границах его области действия. Применение наследования как техники программирования имеет в объектно-ориентированной разработке ПО очень важное значение. Следовательно, программисты не должны ограничивать его практическое использование из-за каких-то абстрактных принципов. Чтобы соответствовать и принципам (одному — концептуальному, другому — техническому), и потребностям программиста, в C++ делаются две оговорки.

На концептуальном уровне в C++ утверждается, что объект производного класса является объектом базового класса, следовательно, он имеет все данные и методы, определенные в базовом классе. Согласно правилам области действия (в знакомом нам виде — для файла, функции, блока и класса) методы базового класса доступны производному классу.

Однако не следует беспокоиться об этих концептуальных и технических проблемах. Имейте в виду, что когда компилятор не находит метод в спецификации производного класса, он видит его в спецификации базового касса. Позднее вы познакомитесь с правилами области действия и разрешения имен при наследовании.

Во втором случае, когда метод отсутствует в базовом классе, но имеется в производном классе, применяются стандартные правила интерпретации вызова функции. Компилятор находит метод в спецификации производного класса и на

том успокаивается. Если аргументы не соответствуют сигнатуре функции, то появляется синтаксическая ошибка. Если же аргументы совпадают, генерируется соответствующий вызов.

```
a2.payInterest();                    // метод производного класса
```

Аналогичные правила применяются в третьем случае, когда метод переопределяется в производном классе. Компилятор игнорирует связь наследования. Как уже было показано выше, когда целью сообщения является объект базового класса, компилятор игнорирует соответствующий метод базового класса и методы производных классов. Если цель сообщения — получить объект производного класса, компилятор ищет спецификацию производного класса и останавливается, когда находит метод. Метод будет найден, так как он переопределяется в производном классе.

```
a1.withdraw();                       // метод производного класса
```

Когда число фактических аргументов и их типы соответствуют сигнатуре функции, компилятор генерирует вызов функции. Если совпадения нет, выводится синтаксическая ошибка. Компилятор не обращается к базовому классу в поиске лучшего совпадения. Как будет показано ниже, это может быть источником проблем.

Доступ к сервисам базового и производного классов

Обычно производный класс "является" по сути и базовым классом, т. е. каждый объект производного класса имеет все элементы данных и функции базового класса, а также добавленные и переопределенные данные и методы.

Производный класс — клиент базового класса. Это напоминает любой клиентский код C++ с серверными классами. Клиент использует сервис сервера — его элементы данных и функции. Серверный класс не знает о своих клиентских классах, не знает имен клиентов. Это естественно, поскольку функция серверного класса может быть библиотечной, написанной за годы до создания клиента. Клиентский класс должен знать имена своих серверных классов и открытых сервисов, которые он может использовать.

Например, клиент из листинга 13.4 определяет объект класса Account, обозначая имя класса. Клиентский код получает доступ к сервисам Account по их именам.

```
Account a(1000);                          // объект базового класса
a.deposit(300);                           // метод базового класса
cout << " Итоговые балансы\n объекта Account:
     << a.getBal() << endl;
```

В данном примере класс Account не имеет представления о том, что его использует клиент. Как уже говорилось, класс Account разрабатывался за несколько лет до создания клиентов совсем другими программистами.

Аналогично, производный класс использует сервисы базового класса (данные и функции). Базовый класс не знает о производных классах, так как при программировании клиенты в сервере никогда не идентифицируются. Производный класс должен знать имя своего базового класса и имена не являющихся закрытыми сервисов, доступных для использования.

Например, производный класс в листинге 13.4 устанавливает связь наследования с базовым классом Account, указывая имя базового класса после двоеточия.

```
class SavingsAccount : public Account {     // синтаксис производного класса
  double rate;
public:
  . . . } ;                                 // остальная часть SavingsAccount
```

Между связями клиент/сервер при композиции классов (агрегации) и связями наследования (производный/базовый) есть разница. При композиции для получения доступа к сервисам клиент должен создавать экземпляр серверного объекта. При наследовании производному классу не нужно задавать экземпляр отдельного базового объекта. В определении производного класса достаточно использовать имя базового класса.

При композиции класса объект-контейнер не предоставляет своим клиентам сервис собственных компонентов. Он предоставляет только свой собственный сервис, явно определяемый в его интерфейсе. Например, класс Point, использовавшийся как компонент класса Rectangle, имеет общедоступные методы set(), get() и move().

```
class Point {
  x, y;                                        // закрытые координаты
    public:
      Point (int a, int b;)                    // обобщенный конструктор
          { x = a; y = b; }
      void set (int a, int b)                  // функция-модификатор
          { x = a; y = b; }
      void move (int a, int b)                 // функция-модификатор
          { x += a; y += b; }
      void get (int& a, int& b) const          // функция-селектор
          { a = x; b = y; } } ;
```

Это не означает, что класс Rectangle, содержащий элемент данных Point, может предоставить своим клиентам те же сервисы. Пример клиента:

```
Point p1(20,40), p2(70,90);          // верхний левый, нижний правый углы
Rectangle rec(p1,p2,4);              // составной объект: клиент Point
rec.set(30,40);                      // это не имеет смысла
rec.move(10,20);                     // это нормально: в чем разница?
```

Разница между методами set() и move() здесь в том, что класс Rectangle не беспокоится о реализации функции-члена set(), но определяет, что означает метод move() в контексте класса Rectangle.

```
class Rectangle {
  Point pt1, pt2;                          // верхний левый, нижний правый углы
  int thickness;                           // толщина границы прямоугольника
public:
  Rectangle (const Point& p1, const Point& p2, int width=1);
  void move(int a, int b);                 // перемещение обоих точек
  void setThickness(int width = 1);        // изменить толщину линии
  bool pointIn(const Point& pt) const;     // точка в прямоугольнике?
    . . . . } ;                            // остальная часть Rectangle
```

Между тем производный класс предлагает своим клиентам сервис базового класса. Разработчику производного класса для этого ничего не нужно делать. Рассмотрим, например, класс SavingsAccount из листинга 13.4.

```
class SavingsAccount : public Account {      // еще один производный класс
  double rate;                               // дополнительные компоненты
public:
  SavingsAccount(double initBalance)
  { balance = initBalance; rate = 6.0; }     // для сберегательных счетов
  void payInterest()                         // для сберегательных счетов
  { balance += balance * rate / 365 / 100; } } ;
```

Клиент данного класса может определять объекты типа SavingsAccount и передавать им сообщения payInterest(). Если же обратиться к клиенту из листинга 13.4, можно увидеть гораздо больше, чем просто передачу этого сообщения.

```
SavingsAccount a2(1000);                          // объект производного класса
a2.deposit(100);                                  // метод базового класса
a2.withdraw(200);                                 // метод базового класса
a2.payInterest();                                 // метод производного класса
cout << " объект SavingsAccount: " << a2.getBal() << endl;
```

Сервисы deposit(), withdraw() и getBal(), используемые в клиенте, не перечисляются в производном классе SavingsAccount. Они перечисляются только в базовом классе Account. Для компилятора это не проблема. Он легко следует по цепочке наследования в определении класса и находит данные функции-члены в базовом классе. Что же делать программисту, работающему над клиентом? Откуда ему знать, что эти сервисы доступны для объектов, определяемых в клиенте? Ему нужно сделать то же, что и компилятору: пройтись по цепочке наследования в определениях классов.

Программисту, использующему сервисы SavingsAccount, следует найти средства Account и понять, что они доступны для объектов SavingsAccount. В листинге 13.4 эти определения классов расположены вместе. В крупных системах со сложной иерархией наследования (когда производный класс используется как базовый для другого класса и т. д.) это не всегда возможно. Поиск списка средств, предоставляемых производным классом, становится для программиста трудной задачей. Описания производного класса уже недостаточно — приходится искать их в другом месте.

Тем самым усложняется программа, возникают ошибки, которые трудно обнаружить и нелегко исправить. Снова встает вопрос о соответствии наследования принципам объектно-ориентированного программирования. Наследование удобно для программиста, разрабатывающего классы в его иерархии. Это метод для повторного использования разработанных фрагментов ПО и уменьшения объема исходного кода.

Что касается разработчика клиента, то два разных класса — SavingsAccount и CheckingAccount — являются неплохим техническим решением. Они связывают родственные данные и функции. Попытка передать сообщение неверному классу помечается компилятором как ошибка. Что добавляет к этому наследование? Данные и методы, общие для обоих классов, нужно реализовывать только один раз, а изменения в базовом классе распространяются на все производные классы автоматически. Такой подход очень удобен при реализации серверных классов.

С другой стороны, наследование затрудняет изучение свойств сервера. Некоторые библиотеки языка C++ снабжают свои классы большим числом сервисов (более 100). Эти сервисы распространяются на пять или более уровней наследования. Чтобы понять работу библиотечного класса, нужно исследовать все уровни наследования. А это непростая задача, поскольку сама иерархия и доступные сервисы меняются от одной версии библиотеки к другой. Таким образом, вам надо совершенствовать свои знания, чтобы быть в курсе изменений. Программирование на C++ — нескучное занятие, особенно когда без всякой меры используется наследование.

В отличие от унаследованных, переопределенные средства непосредственно доступны в списке сервисов производного класса. Их не нужно нигде искать. Обычно они делают то же, что и сервисы, определенные в базовом классе, но более эффективно или с применением несколько других алгоритмов или данных.

В примере наследования, приведенном в листинге 13.4, в производном классе CheckingAccount переопределялась функция-член withdraw() из базового класса Account.

Переопределенная функция использует данные (элемент данных fee), доступные только в производном, но не в базовом классе. Обычно это происходит, потому что в других производных классах (в нашем примере SavingsAccount) такие данные не используются. Если же они там необходимы (в примере все производные классы применяют базовый элемент данных balance), то элемент данных следует включить в базовый класс (как в программе из листинга 13.4).

Применение дополнительных данных в функциях-членах, переопределенных в производном классе,— популярный и распространенный, но не обязательный прием.

Объекты производных классов можно рассматривать как сумму частей производного класса (его компонентов private, protected и public) и частей базового класса (компонентов private, protected и public этого класса). Память, распределяемая для объекта производного класса, также представляет собой сумму областей памяти для частей базового и производного классов.

Например, на нашей машине размер объектов класса Account равен 8 байт, а размеры объектов CheckingAccount и SavingsAccount составляют 16 байт каждый. Если типы данных, используемых как элементы данных, нужно выравнивать в памяти, может потребоваться дополнительное пространство.

Клиент производного объекта вызывает общедоступные сервисы базового класса, используя производный объект, как будто данные сервисы находятся в его части public. Например, объект CheckingAccount отвечает на сообщения deposit() и getBal(), как если бы они были определены в классе CheckingAccount. Клиент не знает о различиях, и ему не нужно о них знать.

Компоненты базового класса не имеют доступа к средствам, добавленным или переопределенным в производных классах. Например, у класса Account нет доступа к закрытому элементу данных rate и общедоступной функции-члену payInterest(), определяемым в классе SavingsAccount. Следующая запись бессмысленна.

```
Account a(1000); a.payInterest();          // синтаксическая ошибка
```

Такие синтаксические правила расширяют понятие, согласно которому объект производного класса является объектом базового класса, плюс что-то еще.

Что касается объектов базового класса, то они не могут ничего знать о сервисе другого класса, даже если это производный от них класс. Все равно класс другой. Объект базового класса не может отвечать на сообщения, не описанные в спецификации.

Аналогично, определение функции или класса как "дружественных" классу Account не предоставляет этой функции прямой доступ к отличным от public компонентам производных классов CheckingAccount и SavingsAccount.

Доступ к базовым компонентам объекта производного класса

Компоненты и "друзья" производного класса получают доступ ко всем элементам данных и его функциям-членам. Кроме того, они имеют некоторый доступ к элементам данных и функциям-членам базового класса. Они могут обращаться только к компонентам public и protected, но не к закрытым данным и функциям-членам базового класса. Они также не получают доступ к компонентам других классов, производных от того же базового класса.

Базовый класс имеет три вида клиентов (три области доступа). Во внутренней области находятся самые большие права доступа к элементам данных и функциям. Их получают функции-члены класса и его "друзья". Они имеют доступ к элементам данных и функциям private, protected и public. Эти права даются им по определению, поскольку они объявлены в границах фигурных скобок класса

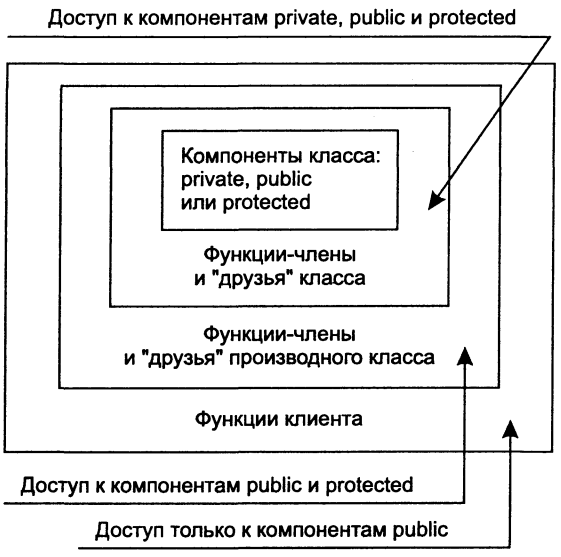


Рис. 13.5. *Области доступа из собственных компонентов и "друзей" класса, из производных классов и из клиента*

как компонентные или "дружественные". В средней области — функции-члены производных классов и их "друзей". Они могут обращаться к компонентам public и protected класса, но не к закрытым компонентам. Суть этого доступа — в объявлении данного класса (прямо или косвенно) как базового в определении класса.

Внешняя область доступа — то, что называется клиентом. Как известно, клиенты имеют доступ только к функциям-членам и данным класса, объявленным как public. Клиент получает доступ к сервисам класса, используя объект класса в качестве адресата сообщения. Этот объект можно сделать доступным для клиента тремя разными способами. Например, создать с помощью определения, динамически в распределяемой области памяти или получить как параметр функции (либо как собственно объект, либо как ссылку или указатель). Эти связи между классом и тремя областями доступа показаны на рис. 13.5.

Обратите внимание, что только во внешней области действия клиент обращается к элементам данных и функциям через отдельный серверный объект. В двух других областях он относится к элементам данных и функциям того же объекта. Во внутренней области это объект базового класса, а в средней — объект производного класса.

То, что происходит в средней области, зависит от режима создания производного класса. Компоненты базового класса способны изменять в объектах производного класса свой статус доступа. То, что было объявлено как public в базовом классе, может превратиться в private в классе производного объекта.

Наследование компонентов public

Каждый класс может наследовать через режим private, public или protected. Режим определяет статус доступа в производном классе к элементам базового класса. При наследовании public статус доступа сохраняется. Компоненты private, public или protected базового класса остаются в объекте производного класса private, public или protected. Это случай с наименьшими ограничениями — ничего не меняется.

Следовательно, методы производного класса могут обращаться к базовым компонентам public или protected производного объекта. Эта связь показана на рис. 13.6. Здесь демонстрируется объект производного класса, состоящий из базовых и производных частей. Каждая часть содержит компоненты private, public или protected. Показан также клиент, использующий объект производного класса как свой сервер. Клиент может обращаться к сервисам public базового класса (данным и функциям) и сервисам public производного класса. Для клиента объект производного класса выглядит как совокупность общедоступных средств, определенных в базовом и производном классах.

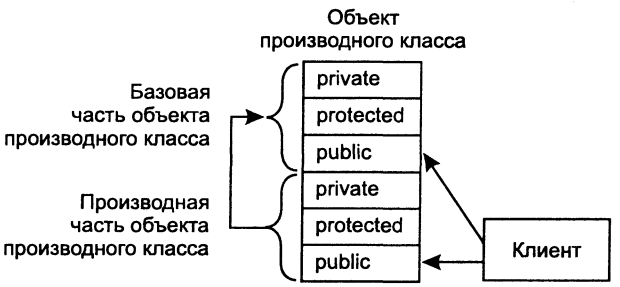


Рис. 13.6. *Доступ к сервисам базового и производного классов из объекта производного класса и из клиента при режиме наследования public*

Объект производного класса может обращаться только к компонентам private и protected, унаследованным из базового класса. Для доступа к собственным закрытым компонентам, унаследованным из базового класса, следует использовать функции доступа базового класса. На первый взгляд это неразумно. Объект производного класса не имеет доступа к собственным компонентам! Ничего подобного нам раньше не встречалось.

С другой стороны, производный класс является клиентом базового класса. Базовый класс может иметь элементы (особенно данные), архитектура которых со временем меняется. Если сделать эти элементы доступными для производных классов, то их потребуется и модифицировать. При доступе через отличные от private функции производные классы защищены от влияния изменений в базовых классах. По этой же причине элементы данных объявляются закрытыми, а функции-члены — общедоступными.

Листинг 13.5. Доступ к компонентам классов Base и Derived в производном объекте для класса Derived и для клиента при наследовании в режиме public

```
#include <iostream>
using namespace std;

class Base {
   private: int privB;                  // доступ только из Base
   protected: int protB;                // доступ из Base и Derived
   public: void publB()                 // доступ из Base, Derived, Client
   { privB = 0; protB = 0; } } ;        // OK для доступа к собственным данным

class Derived : public Base {           // режим наследования public
   private: int privD;
   protected: int protD;
   public: void publD()
   {  privD = 0; protD = 0;             // OK для доступа к собственным данным
      protB = 0;                        // OK для доступа к унаследованным компонентам
//    privB = 0;                        // нет доступа к унаследованным компонентам
      } } ;

class Client {
public: Client()                        // конструктор класса Client
{  Derived d;                           // объект производного класса
   d.publD();                           // OK для доступа к сервисам public
   d.publB();                           // OK для доступа к сервисам public класса Base
   // d.privD = d.protD = 0;            // нет доступа к отличным от public сервисам
   // d.privB=d.protB=0;                // нет доступа к отличным от public сервисам
    };

int main()
{  Client c;                            // создает объект, выполняет программу
   return 0;
   }
```

В листинге 13.5 показана связь между объектом производного класса и его собственными компонентами. Здесь класс Derived является производным от класса Base в режиме public. Класс Base, как и Derived, имеет компоненты private, public и protected. Производный класс в методе publD() может обращаться к собственным компонентам privD и protD. Кроме того, он может относиться к унаследованным компонентам public и protected класса Base — protB и publB(). Между тем попытка класса Derived обратиться к закрытым компонентам privB, унаследованным из класса Base, была бы синтаксической ошибкой,

хотя в объекте Derived распределяется память для этого компонента. Класс Client создает объект класса Derived в своем конструкторе и обращается к общедоступным сервисам, определенным в классе Derived (publD()) и в классе Base (publD()). Он не может обращаться к отличным от public компонентам классов Base и Derived. Поскольку здесь демонстрируются права доступа к средствам Base и Derived, программе не нужно выдавать никакого результата. Она генерирует только сообщения компилятора об ошибках.

Наследование public — наиболее естественный режим наследования, так как при этом сохраняется связь типа "является видом" между классами. При создании производного класса в таком режиме объекты предлагают клиенту все общедоступные средства, присутствующие в базовом объекте, и добавляют дополнительные. Возможности дальнейшего наследования также не ограничиваются.



Внимание При создании производного класса в режиме public унаследованные компоненты базового класса сохраняют в объектах производного класса свой статус доступа (private, protected или public). Все общедоступные (public) сервисы, определенные в производном классе и унаследованные из базового, доступны клиенту. Это самый естественный режим наследования.

Листинг 13.6 показывает более крупный пример использования наследования. Для доступа к компонентам x и y базовый класс Point предлагает два сервиса public — set() и get(). Производный класс VisiblePoint добавляет к этим средствам элемент данных visible и функции-члены show(), hide() и retrieve(). Метод show() устанавливает элемент данных visible в 1. В результате точка будет отображаться графической программой. Наследование происходит в режиме public. Вместо числовых значений для отображаемых и скрытых точек лучше было бы использовать перечисление, но для компактности примера выбраны именно числа.

Листинг 13.6. Доступ к базовым компонентам в производном объекте при наследовании в режиме public

```
#include <iostream>
using namespace std;

class Point {                              // базовый класс
  int x, y;                                // закрытые данные базового класса
public:
  void set (int xi, int yi)
    { x = xi; y = yi; }
  void get (int &xp, int &yp) const        // общедоступные методы базового класса
    { xp = x; yp = y; } } ;

class VisiblePoint : public Point {        // двоеточие перед public
  int visible;

public:                                    // двоеточие после public
  void show()
    { visible = 1; }

  void hide()
    { visible = 0; }

  void retrieve(int &xp, int &yp, int &vp) const;
    { xp = x; yp = y;                      // синтаксическая ошибка: закомментируйте ее!
      get(xp,yp);                          // доступ к методу базового класса
      vp = visible; } } ;                  // производные закрытые данные: OK
```

```
int main()
{
  VisiblePoint a,b; int x,y,z;          // определение двух производных методов
  a.set(0,0); b.set(20,40);             // вызов функции базового класса
  a.hide(); b.show();                   // вызов метода public производного класса
  a.get(x,y);                           // вызов функции public базового класса
  b.retrieve(x,y,z);                    // вызов метода public производного класса
  cout << " Координаты точки: x=" << x <<" y=" << y << endl;
  cout << " Видимость точки: visible=" << z << endl;
  return 0;
  }
```

Общедоступные функции-члены базового класса `Point::set()` и `Point::get()` доступны в коде клиента таким же образом, как общедоступные методы `VisiblePoint`. Любой объект `VisiblePoint` может предоставить такие возможности своим клиентам. Закрытые элементы данных `Point::x` и `Point::y` в классе `VisiblePoint` недоступны. При попытке выполнить программу появится синтаксическая ошибка в первой строке функции-члена `retrieve()`. Избавиться от нее можно двумя способами. Первый состоит в том, чтобы объявить элемент данных `Point` как `protected`. Если бы он был таковым в классе `Point`, то функция `retrieve()` могла бы обращаться к нему в классе `VisiblePoint`. Тем не менее, в клиенте он будет недоступен. Второй способ заключается в использовании в функциях-членах `VisiblePoint` функций доступа `Point`, допускающих обращение к закрытым данным базового класса. Этот способ демонстрируется во второй строке функции `retrieve()`. Если закомментировать первую строку `retrieve()` (с синтаксической ошибкой), то программа будет работать и даст результаты, представленные на рис. 13.7.

```
Координаты точки: x=20 y=40
Видимость точки: visible=1
```

Рис. 13.7. *Результаты программы из листинга 13.6*

Предпочтительнее использовать первый вариант, когда данные базового класса объявляются `protected`, а не `private`. При этом в базовом классе применяется меньше функций доступа и упрощается исходный код производного класса. Программисты, предпочитающие использовать функции доступа, могут возразить, что прямой доступ к данным `protected` базового класса из производного класса представляет то же нарушение инкапсуляции, как и прямой доступ к элементам данных `public` при взаимодействии клиента и сервера. Как уже отмечалось выше, они правы, однако в данном случае проблема незначительна. Если вы почувствуете, что могут возникнуть серьезные трудности, сделайте данные базового класса закрытыми и используйте в производных классах функции доступа. В противном случае просто не беспокойтесь об инкапсуляции больше, чем это действительно необходимо.



Внимание Объект производного класса не может обращаться к своим унаследованным компонентам, которые являются закрытыми в базовом классе, хотя они "принадлежат" объекту производного класса. Для обращения к этим элементам данных производного класса используйте функции доступа базового класса или объявите эти компоненты в базовом классе как `protected`. Для производного класса компоненты `protected` базового класса ничуть не хуже, чем компоненты `public` (т. е. доступны в той же степени). Для клиента они приравниваются по доступу к `private` (т. е. он не может к ним обращаться).

Наследование в режиме protected

Такое наследование представляет собой механизм, ограничивающий доступ к сервисам базового класса. Компоненты public и protected, наследуемые из базового класса, становятся компонентами protected в объекте производного класса.

Сервисы базового класса доступны для дальнейшего наследования и используются в методах производного класса. Однако клиент не может обращаться через объект производного класса к сервисам public базового класса, поскольку теперь они являются защищенными (protected). На рис. 13.8 показаны изменения в отношении режима наследования protected. То, что было public в базовом классе, становится protected в производном классе. Пунктирная линия показывает, что доступ для клиента к этой части объекта производного класса запрещен.

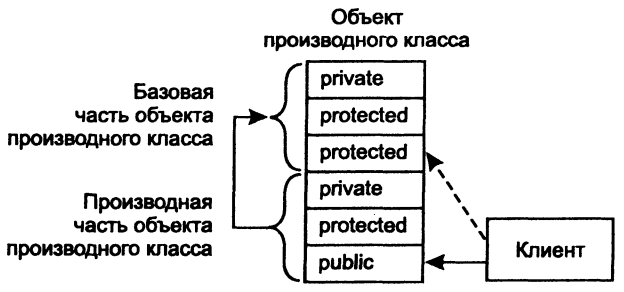


Рис. 13.8. *Доступ к компонентам базового класса и производного класса из объекта производного класса и из клиента при наследовании в режиме protected*

Листинг 13.7 демонстрирует абстрактный пример из листинга 13.5, когда режим наследования public заменяется на protected. Этот пример иллюстрирует соотношения между объектом производного класса и его собственными компонентами, а также между данным объектом и клиентом.

Листинг 13.7. Доступ к компонентам Base и Derived объекта Derived для класса Derived и клиента при режиме наследования protected

```
#include <iostream>
using namespace std;

class Base {
   private: int privB;                  // доступ только из Base
   protected: int protB;                // доступ из Base и Derived
   public: void publB()                 // доступ из Base, Derived, Client
   { privB = 0; protB = 0; } } ;        // OK для доступа к собственным данным

class Derived : protected Base {        // режим наследования protected
   private: int privD;
   protected: int protD;
   public: void publD()
   {  privD = 0; protD = 0;             // OK для доступа к собственным данным
      protB = 0;                        // OK для доступа к унаследованным компонентам
//    privB = 0;                        // нет доступа к унаследованным компонентам
       } } ;

class Client {                          // код клиента
public:
   Client()
   { Derived d; Base b;                 // объекты производного и базового классов
     d.publD();                         // часть public производного класса: OK
//   d.publB();                         // нет доступа к сервисам public класса Base
//   d.privD = d.protD = 0;             // нет доступа к отличным от public сервисам Derived
//   d.privB=d.protB=0;                 // нет доступа к отличным от public сервисам Base
     b.publB(); }                       // объект Base: часть public, OK
        }
```

```
int main()
{  Client c;                              // создает объект, выполнят программу
   return 0;
   }
```

Вспомним вызов функции-члена publB() класса Base с производным объектом в качестве получателя сообщения — d.publB() (в предыдущем примере из листинга 13.5). В листинге 13.7 это синтаксическая ошибка. Заметим, что доступ к компонентам Base отвергается только тогда, когда он осуществляется через объект класса Derived. В конце применяемого по умолчанию конструктора Client() вызываемая функция-член publB() использует в качестве целевого объекта объект b класса Base. Компонент public доступен клиентам этого класса. Данный сервис недоступен клиентам класса Derived.

Листинг 13.8. Доступ к базовым компонентам в производном объекте при наследовании в режиме protected

```
#include <iostream>
using namespace std;

class Point {                          // базовый класс
protected:
   int x, y;                           // закрытые данные базового класса
public:
   void set (int xi, int yi)
      { x = xi; y = yi; }
   void get (int &xp, int &yp) const         // общедоступные методы базового класса
      { xp = x; yp = y; } } ;

class VisiblePoint : protected Point {       // наследование protected
   int visible;

public:
   void show()
      { visible = 1; }

   void hide()
      { visible = 0; }

   void retrieve(int &xp, int &yp, int &vp) const;
      { xp = x; yp = y;                      // доступ к данным protected: OK
//      get(xp,yp);                          // не будем усложнять
        vp = visible; }

   void initialize(int xp, int yp, int vp)   // новый сервис public
      { x = xp; y = yp;                      // доступ к данным protected базового класса
        visible = vp; } } ;                  // доступ к данным private производного класса

int main()
{  VisiblePoint a,b; int x,y,z;            // определение двух производных методов
   b.initialize(20,40,1);                  // инициализация производного объекта
//  a.set(0,0); b.set(20,40);              // теперь это синтаксическая ошибка
   a.hide(); b.show();                     // вызов метода public производного класса: OK
//  a.get(x,y);                            // это синтаксическая ошибка
   b.retrieve(x,y,z);                      // вызов метода public производного класса
   cout << " Координаты точки: x=" << x << " y=" << y << endl;
   cout << " Видимость точки: visible=" << z << endl;
   return 0;
   }
```

В листинге 13.8 показан пример из листинга 13.6, где режим наследования public заменен на protected. Базовый класс Point предлагает те же сервисы set() и get(), но клиент производного класса VisiblePoint не может их использовать — это защищенные объекты VisiblePoint. Попытка сделать это в конструкторе Client() дает синтаксическую ошибку. Чтобы разрешить проблему, в класс VisiblePoint добавлен новый сервис initialize(), обращающийся к унаследованным элементам данных x и y вместо get() и set(). Обратите внимание, что теперь производный класс без проблем обращается к базовому классу, поскольку данные базового класса объявлены как protected. В функции-члене retrieve() производного класса закомментирован вызов базовой функции get().

Если закомментировать две строки с синтаксическими ошибками, программа будет работать и давать тот же результат, что и программа из листинга 13.6 (см. рис. 13.7).

Остается надеяться, что наследование protected понравилось вам больше, чем public. Наследование public — это способ добавления сервисов в дополнение к предлагаемым базовым классам или замены некоторых сервисов (без изменения имени) на что-то более полезное для клиента производного класса. Во всех примерах наследования public между производными и базовыми объектами соблюдается отношение "является видом". Для клиента объект SavingsAccount "является видом" объекта Account со средствами начисления процентов. Объект VisiblePoint (видимая точка) представляет собой вид объекта Point (точка) с добавленным свойством "отображаемая/скрытая".

При наследовании protected все происходит иначе. Это техника для быстрого создания класса, использующего отличные от public сервисы базового класса (элементы данных x и y в листинге 13.8). Однако в этом случае клиентам не предоставляются сервисы public базового класса (методы set() и get() в листинге 13.8). Им предлагается другой набор сервисов (метод initialize() в листинге 13.8), которые, по тем или иным причинам, больше подходят для клиента.

В листинге 13.8 объект VisiblePoint не является объектом Point. Объекты Point предлагают клиентам методы set() и get(), а объект VisiblePoint — нет.

Еще один популярный пример использования наследования protected — создание стекового класса, предоставляющего клиентам доступ к конечному элементу стека. Он является производным от класса массива, который предоставляет доступ к любому компоненту. Используя наследование, разработчик может запретить клиентам работать с массивом. Стек предлагает методы push() и pop() для доступа к вершине стека.

В начале данной главы уже отмечалось важное различие между наследованием и композицией классов. Производные классы предоставляют своим клиентам все сервисы public базового класса. Составной класс не дает клиентам сервисов свои компоненты, если эти сервисы не поддерживают методы составного класса. (В приведенных примерах класс Rectangle предусматривал сервис move().)

Если нужно закрыть клиенту доступ к некоторым из существующих сервисов, не используйте наследование. Вместо него применяйте композицию классов.

В листинге 13.9 показан пример из листинга 13.8, но класс VisiblePoint теперь является компонентом данных класса Point, а не наследником Point в режиме protected. Результат примера тот же, что и на рис. 13.7.

Сервисы set() и get() базового класса Point скрываются от клиента из-за композиции классов. Элементы данных Point оказываются скрытыми внутри объекта VisiblePoint и недоступны клиенту. В результате клиент не сможет делать с объектом VisiblePoint то, что он мог делать с объектом Point — перемещать по экрану безотносительно к видимости. Вместо этого класс VisiblePoint предоставляет клиенту собственный интерфейс — функции-члены initialize() и retrieve(), требующие, чтобы клиент работал с видимыми объектами.

Все это применимо к любой ситуации, когда нужно спроектировать для обслуживания клиентов новый класс. Более того, у вас есть класс, который желательно

использовать в данном проекте. Если клиенту нужны все сервисы существующего класса плюс еще что-то, лучше применять наследование из этого класса в режиме public. Для унаследованных и добавленных сервисов клиент будет использовать объекты производного класса. Если сервисы существующего класса будет использовать не клиент, а новый класс, не стоит применять наследование, даже в режиме protected. Лучше прибегнуть к композиции классов (см. главу 12).

Листинг 13.9. Использование композиции классов вместо наследования

```
#include <iostream>
using namespace std;

class Point {                                          // компонентный класс
private:
  int x, y;                                            // закрытые данные
public:
  void set (int xi, int yi)
    { x = xi; y = yi; }
  void get (int &xp, int &yp) const                    // общедоступные методы базового класса
    { xp = x; yp = y; } } ;

class VisiblePoint {                                   // нет наследования, композиция
  Point pt;                                            // закрытый компонент
  int visible;

public:
  void show()                                          // новый сервис для клиента
    { visible = 1; }                                   // новый сервис для клиента

  void hide()
    { visible = 0; }

  void retrieve(int &xp, int &yp, int &vp) const;      // замена
    { pt.get(xp,yp);                                   // сервисы скрыты от клиента
      vp = visible; }

  void initialize(int xp, int yp, int vp)              // замена
    { pt.set(xp,yp);                                   // сервисы скрыты от клиента
      visible = vp; } } ;                              // аналогично скрытым частным данным

int main()
{
  VisiblePoint b; int x, y, z;                         // определение составного объекта
  b.initialize(20,40,1);                               // сервис составного класса
  b.show();                                            // сервис составного класса
  b.retrieve(x,y,z);                                   // сервис составного класса
  cout << " Координаты точки: x=" << x << " y=" << y << endl;
  cout << " Видимость точки: visible=" << z << endl;
  return 0;
  }
```

Наследование в режиме protected может быть полезным в ситуации, когда разрабатывается класс (или семейство классов) для обслуживания клиентов. Советуем строить этот класс поэтапно.

Например, клиенту требуется класс D1 и нужно получить производный класс D1 из D, который, в свою очередь, является производным от класса B. Применяя наследование в режиме protected для создания класса D, производного от B, и класса D1, производного от D, можно построить класс D1, использующий все сервисы

public и protected класса D. Эти сервисы включают в себя сервисы public и protected класса B. Клиент D1 не будет использовать сервисы public классов B и D, которые скрываются от клиента благодаря режиму наследования protected.

Другими словами, наследование protected предоставляет способ для ограничения доступа клиентов к сервисам public базового класса. При этом не ограничивается доступ к этим сервисам из производного класса и продолжается дальнейшее наследование.



Внимание Наследование в режиме protected скрывает от клиента сервисы базового класса, используемые объектом производного класса. Тем самым искажается отношение "является видом". Если оно не важно, используйте вместо наследования protected композицию классов. Если же чувствуете, что следует применить наследование, лучше использовать режим public. (Хотя это предвзятая точка зрения.)

Наследование в режиме private

Наследование в режиме private представляет собой технику ограничения доступа к базовым сервисам не только для клиентов производных классов, но и для производных классов от данных производных классов.

Когда базовый класс используется как private, все его компоненты public и protected становятся в объекте производного класса компонентами private. Они недоступны для клиентов производных классов, а также для методов классов, производных от производного класса. Их могут использовать методы производного класса.

В этом состоит важное отличие использования базового класса как private от других режимов наследования. При наследовании protected и public правила доступа транзитивны. Если производный класс использует базовый класс для дальнейшего наследования (protected или public), то производный класс имеет те же права доступа к компонентам базового класса, что и класс, непосредственно полученный из базового. Если же наследование происходит в режиме private и производный класс используется для дальнейшего наследования, то его потомки не будут иметь доступа ни к каким компонентам базового класса. К компонентам protected и public базового класса может обращаться только класс, непосредственно производный от базового (дочерний). Тем самым разработчик производного класса предотвращает использование сервисов базового класса в дальнейшем наследовании.

Как и в случае наследования protected, общедоступный интерфейс (public) базового класса (данные и методы) не является частью интерфейса производного класса. Для клиента все это закрыто (private).

Данные связи показаны на рис. 13.9. При наследовании в режиме private объект класса, производного от производного класса, не имеет доступа к своим собственным компонентам, унаследованным от базового класса.

Листинг 13.10 снова показывает небольшой абстрактный пример. На этот раз используется наследование private. Что же касается прав доступа производного объекта к своей базовой части, то они здесь такие же, что и в предыдущем примере (с наследованием в режиме protected).

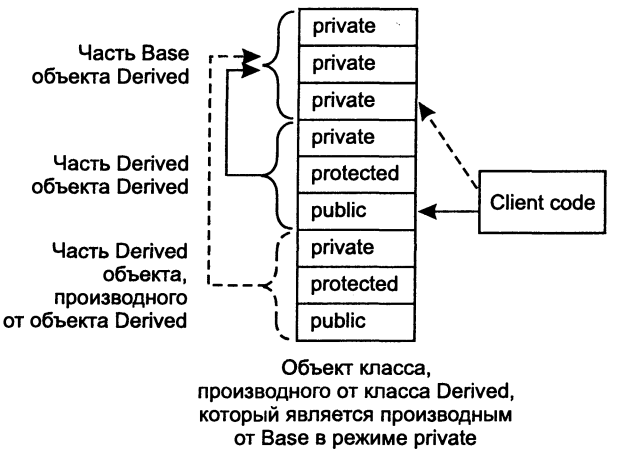


Рис. 13.9. *Доступ к базовым компонентам из объекта производного класса при наследовании private*

Листинг 13.10. Доступ к компонентам Base в иерархии наследования, когда класс Derived наследует из класса Base в режиме private

```
#include <iostream>
using namespace std;

class Base {
   private: int privB;                    // доступ только из Base
   protected: int protB;                  // доступ из Base и Derived
   public: void publB()                   // доступ из Base и Derived
   { privB = 0; protB = 0; } } ;          // OK для доступа к собственным данным

class Derived : private Base {            // режим наследования private
   private: int privD;
   protected: int protD;
   public: void publD()
   { privD = 0; protD = 0;                // OK для доступа к собственным данным
     protB = 0;                           // OK для доступа к унаследованным компонентам
//   privB = 0;                           // нет доступа к унаследованным компонентам

class Derived1 : public Derived {         // класс, производный от Derived
public: void publDD()
{  // privD = 0                           // нет доступа к "базовым" данным private
   protD = 0;                             // OK для доступа к "базовым" данным protected
   publD();                               // OK для доступа к "базовым" данным public
//   protB = 0;                           // нет доступа к любой части "закрытой базы"
     publB();                             // нет доступа к любой части "закрытой базы"
} } ;

class Client {
public:
   Client()
{  Derived d; Base b;                     // объекты производного и базового классов
   d.publD();                             // часть public класса Derived: OK
//   d.publB();                           // часть public Base класса Derived: не OK
//   d.privD = d.protD = 0;               // отличная от public часть Derived: не OK
//   d.privB=d.protB=0;                   // отличная от public часть Base в Derived: не OK
   b.publB(); }                           // часть public Base объекта Base: OK
    }

int main()
{  Client c;                              // создает объект, выполняет программу
   return 0;
   }
```

Чтобы программу можно было скомпилировать, здесь закомментированы несколько нарушающих правила строк. Производный класс может обращаться ко всем отличным от private компонентам базового класса. Это не зависит от режима наследования. Аналогично, класс Derived1 может обращаться ко всем не закрытым компонентам собственного "базового" класса (Derived). Это также не связано с режимом наследования. Между тем в режиме private класс Derived1 не может обращаться к компонентам базового класса Base. Что касается клиента, для него наследование private аналогично наследованию protected. Все компоненты базового класса в производном объекте недоступны для клиента.

Наследование private допускает написание новых серверов с помощью уже реализованных элементов. Однако в этом случае подтипы не имеют отношений. Если в режиме private из класса Array создается объект Stack, последний не является объектом Array и не предоставляет клиентам Stack или классам, наследующим из Stack, сервисы Array.

Stack может использовать Array в качестве одного из своих элементов. Применение наследования private или protected — не очень хорошее решение. Используйте композицию классов.

Между тем, некоторые эксперты полагают, что этот режим наследования полезен, поскольку для обращения к закрытым данным он вынуждает использовать в производных классах (как и во всех других клиентах и других классах) методы доступа. Как уже отмечалось выше, это спорный вопрос.

Полиморфизм (о котором рассказывается в следующей главе) доступен только при наследовании public, и это может быть еще одной причиной в пользу режима наследования public. Избегайте наследования private и protected.

Изменение доступа к компонентам базового класса в производном классе

C++ позволяет разработчикам производного класса обойти многочисленные ограничения, налагаемые правилами наследования в режимах protected и private. В производном объекте компонентам базового класса возвращаются те права доступа, которые они ранее предоставляли.

Листинг 13.11 снова показывает пример с закрытым наследованием от класса Base к Derived. В определении класса Derived восстановлен статус protected элемента данных Base::protB. Кроме того, восстановлен статус public функции-члена Base::publB(). Синтаксис будет одинаковым для функции-члена и элемента данных. Права доступа самого класса Derived не изменяются. При любом режиме наследования он может обращаться ко всем отличным от private компонентам базового класса. Клиент же теперь получает доступ к Base::publB(), как если бы класс Derived наследовал от Base в режиме public, а не private.

Листинг 13.11. Пример изменения прав доступа к компонентам Base в классе Derived (наследование в режиме private)

```
#include <iostream>
using namespace std;

class Base {
  private: int privB;                    // доступ только из Base
  protected: int protB;                  // доступ из Base и Derived
  public: void publB()                   // доступ из Base и Derived
  { privB = 0; protB = 0; } } ;          // OK для доступа к собственным данным

class Derived : private Base {           // режим наследования private
  private: int privD;
  protected: int protD;
  protected:
    Base::protB;                         // доступно для последующего наследования
  public:
    Base::publB;                         // доступно для клиента
  public: void publD()
  { privD = 0; protD = 0;                // OK для доступа к собственным данным
    protB = 0;                           // OK для доступа к унаследованным компонентам
//  privB = 0;                           // нет доступа к унаследованным компонентам
    } } ;

class Derived1 : public Derived {        // класс, производный от Derived
public: void publDD()
{ // privD = 0;                          // нет доступа к "базовым" данным private
  protD = 0;                             // OK для доступа к "базовым" данным protected
  publD();                               // OK для доступа к "базовым" данным public
```

```
   publB();                          // OK, если в Derived стал public
   protB=0;                          // OK, если в Derived стал protected
     } } ;

class Client {
public: Client()
{  Derived d; Base b;                // объекты производного и базового классов
   d.publD();                        // часть public класса Derived: OK
   d.publB();                        // OK, если в Derived стала public
//   d.privD = d.protD = 0;          // отличная от public часть Derived: не OK
//   d.privB=d.protB=0;              // отличная от public часть Base в Derived: не OK
   b.publB();                        // часть public Base объекта Base: OK
     } } ;

int main()
{  Client c;                         // создает объект, выполняет программу
   return 0;
   }
```

Благодаря наследованию private можно сделать архитектуру программы весьма запутанной, закрыть доступ к одним компонентам и открыть к другим. Получится головоломка, которую вы можете с гордостью показать своим коллегам и спросить: "Отгадайте, что здесь делается".

На самом деле C++ позволяет не только управлять правами доступа, но и изменять их на нечто другое, отличное от того, что было задано в базовом классе. В производных классах нельзя лишь сделать закрытые компоненты базового класса незакрытыми, установив их, к примеру, в режим protected или public.

Режим наследования по умолчанию

C++ позволяет использовать наследование по умолчанию. В таком режиме предполагается, что программист, занимающийся клиентской частью или сопровождающий программу, обладает достаточными знаниями, чтобы понять происходящее, даже если это явно не указывается.

Листинг 13.12. Пример использования для классов режима наследования по умолчанию

```
class Base {
   private: int privB;                // доступен только из Base
   protected: int protB;              // доступен из Base и Derived
   public: void publB()               // доступен из Base и Derived
   { privB = 0; protB = 0; } } ;      // OK для доступа к собственным данным

class Derived : Base {                // по умолчанию private
   private: int privD;
   protected: int protD;
   public: void publD()
   { privD = 0; protD = 0; protB = 0; } } ;   // OK для доступа

int main()
{  Derived d;                         // объект производного класса
   d.publD();                         // OK для доступа к части public класса Derived
//   d.publB();                       // не OK для доступа к части public класса Base
   return 0;
   }
```

По умолчанию режим наследования для производного класса C++ является закрытым. Если забыть указать режим, то компилятор предполагает, что применяется режим наследования private. Листинг 13.12 показывает пример скелета программы, где программист забыл обозначить режим наследования. В результате клиент не может обращаться к общедоступному методу publB(), унаследованному целевым классом Derived из класса Base.

Все это не так просто, как кажется. Режим по умолчанию при наследовании будет закрытым для производного класса, создаваемого не только с помощью ключевого слова class. Вспомните, что ключевые слова class и struct обозначают одно и то же, за исключением назначаемых по умолчанию прав доступа к элементам данных и функциям. Для класса он будет private, а для структуры — public. В остальном они одинаковы. Можно определять функции-члены в структуре, создавать для них перегруженные функции и назначать аргументы по умолчанию, использовать конструкторы и деструкторы, элементы данных других классов (и структур), списки инициализации и все то, что отличает объектно-ориентированное программирование от процедурного. Вы можете наследовать из структуры и сделать так, чтобы структура наследовала из класса или класс из структуры. Все это законно в C++. Только для производного класса, определенного с помощью ключевого слова struct, режимом по умолчанию будет public, а не private.

Приведем пример класса Derived, определяемого с помощью ключевого слова struct. Поскольку он является производным от своего базового класса в режиме наследования по умолчанию, этим режимом будет public.

Листинг 13.13. Пример использования режима наследования по умолчанию для структур

```
class Base {
   private: int privB;                          // доступен только из Base
   protected: int protB;                        // доступен из Base и Derived
   public: void publB()                         // доступен из Base и Derived
   { privB = 0; protB = 0; } } ;                // OK для доступа к собственным данным

struct Derived : Base {                         // по умолчанию private
   private: int privD;
   protected: int protD;
   public: void publD()
   { privD = 0; protD = 0; protB = 0; } } ;     // OK для доступа

int main()
{  Derived d;                                   // объект производного класса
   d.publD();                                   // OK для доступа к части public класса Derived
   d.publB();                                   // теперь это вполне законно
   return 0;
   }
```

Пример согласуется с правилами, применяемыми в C++ по умолчанию для доступа к компонентам классов. Вспомните, что когда класс определяется с использованием ключевого слова class, по умолчанию права доступа к компонентам класса будут private. Когда класс определяется с помощью ключевого слова struct, по умолчанию права доступа к его компонентам будут public.

Аналогично, при создании производного класса с указанием ключевого слова class режимом наследования является private. Когда класс создается из другого класса с ключевым словом struct, режимом наследования будет public. Вся разница заключается в способе определения производного класса. Базовый класс может определяться с любым ключевым словом — class или struct,— это не влияет на режим наследования производного класса.

Не стоит полагаться на режимы по умолчанию.

Правила области действия и разрешение имен при наследовании

При наследовании области действия классов C++ можно рассматривать как вложенные. С этой точки зрения область действия производного класса "вкладывается" в область действия своего базового класса.

Согласно общей теории вложенных областей действия, то, что определяется во внутренней области, будет невидимым во внешней (глобальной). И наоборот, то, что определяется во внешней области, видимо во внутренней (локальной). В следующем примере переменная x определяется во внешней области действия функции, а переменная y — во внутреннем блоке. Можно обращаться к переменной x во внутреннем блоке, однако бесполезно пытаться использовать переменную y во внешней области действия.

```
void foo()
{ int x;        // внешняя область действия: эквивалент базового класса
  { int y;      // внутренняя область действия: эквивалент производного класса
    x = 0; }    // OK для доступа к имени из внешней области
  y = 0; }      // синтаксическая ошибка: внутренняя область извне невидима
```

В данном примере внешняя область действия играет роль базового класса и его компонентов, а внутренняя представляет производный класс и его компоненты. Из производного класса можно обращаться к компонентам базового класса, но не наоборот: к компонентам производного класса из базового класса доступа нет.

Следовательно, компоненты производного класса невидимы в области действия базового класса. Перед написанием производного класса базовый класс должен проектироваться, реализовываться и компилироваться. Таким образом, вполне естественно, что функции-члены базового класса не могут обращаться к элементам данных или функциям производного класса.

Компоненты базового класса находятся во внешней области действия, и поэтому видимы для методов производного класса. Это понятно, так как производный класс "является видом" объекта базового класса и содержит все элементы данных и функции, имеющиеся в базовом классе. С этой точки зрения модель области действия в связях между базовым и производными классами не особенно полезна. Однако она нужна, если производный и базовый классы используют одни и те же имена. В разных языках для разрешения подобных конфликтов имен применяются разные правила. Модель вложенных областей действия, используемая в C++, может помочь в развитии "программистской интуиции" при написании программного обеспечения на C++.

Область действия производного класса вложена в область действия базового класса. Это означает, что имена производного класса скрывают в нем имена базового класса. Аналогично, имена производного класса скрывают в клиенте имена базового класса, существующие в производном классе. Вы должны помнить об этом правиле. Если в производном и базовом классах применяется одно имя, то используется имя производного, а не базового класса.

Поясним это правило. Если имя без операции области действия обнаруживается в функции-члене производного класса, компилятор пытается использовать его как локальное для данной функции. В следующем примере действуют четыре переменные. Все они носят имя x и имеют одинаковый тип, но это не важно. Они могут быть и разных типов, и некоторые из имен могут даже обозначать функцию. Общее правило все равно будет действовать.

```
int x;                  // внешняя область действия: ее скрывает класс или функция
class Base {
  protected: int x;  // имя базового класса скрывает глобальное имя
} ;
```

```
class Derived : public Base {
  int x;              // имя производного класса скрывает имя базового
public:
  void foo()
  { int x;
    x = 0; } } ;      // локальная переменная скрывает все другие имена

class Client {
public:
  Client()
  { Derived d;
    d.foo(); } } ;   // использование объекта d как получателя сообщения

int main()
{ Client c;           // определение объекта, выполнение программы
  return 0; }
```

В этом примере видно, что используются локальная переменная в функции-члене `foo()` класса `Derived`, элемент данных класса `Derived`, элемент данных класса `Base` и глобальная переменная в области действия файла. Оператор `x = 0;` в `Derived::foo()` устанавливает локальную переменную `x` в значение `0`. Элемент данных производного класса `Derived::x`, элемент данных базового класса `Base::x` и глобальное имя `x` скрываются этим локальным именем, так как оно определено во внутренней вложенной области действия.

Закомментируйте определение переменной `x` в методе `foo()`. Тогда в операторе `x = 0;` при разрешении имен используется не локальная переменная — ее имя найдено не будет. Если имя не найдено в области действия оператора (в данном случае это компонентная функция производного класса), то компилятор просматривает область действия производного класса и ищет имя среди элементов данных и функций. В приведенном примере, если бы отсутствовала локальная переменная `x` в функции `Derived::foo()`, использовалось бы имя `Derived::x`. Оно будет устанавливаться в `0` оператором `x = 0;` в функции-члене `Derived::foo()` производного класса.

Если упомянутое в функции имя не найдено и в области действия класса, то компилятор выполняет поиск в базовом классе. Первое имя, которое находит при таком поиске компилятор, будет использоваться для генерации объектного кода. Если бы обе переменные `x` в классе `Derived` отсутствовали (локальная переменная и элемент данных), то элемент данных `Base::x` устанавливался бы в значение `0` оператором `x = 0;`.

Наконец, если компилятор не находит имя ни в одном из базовых классов, он осуществляет поиск в области действия файла (как глобальный объект, определенный в области действия файла, или глобальный объект `extern`, объявленный в области действия файла, но определенный в другом месте). Если при таком процессе имя обнаруживается, оно используется. Если нет, то генерируется синтаксическая ошибка. В приведенном примере, если бы ни класс `Derived`, ни класс `Base` не использовали имя `x`, то в `0` оператором `Derived::foo()` устанавливалась бы глобальная переменная `x`.

Аналогично, если клиент производного класса передает сообщение своему объекту, то компилятор сначала выполняет поиск в производном классе, и только потом просматривает базовый класс. Если базовый класс и один из его производных классов используют одно имя, применяется интерпретация производного класса. Когда имя найдено в производном классе, компилятор даже не ищет его в базовом классе. Имя из производного класса скрывает имя из базового класса, не оставляя ему шансов.

Ниже показан модифицированный пример с двумя классами — `Base` и `Derived`. Здесь заданы две функции `foo()`: одна — общедоступная функция-член класса `Base`, другая — общедоступная функция-член класса `Derived`. Клиент определяет

объект класса Derived и передает ему сообщение foo(). Так как производный класс Derived() определяет функцию-член foo(), вызывается функция-член производного класса. Если класс Derived не обозначает функцию foo(), то компилятор генерирует вызов функции foo() базового класса. Функция базового класса имеет шанс только в том случае, если данное имя не используется в производном классе.

```
class Base {
  protected: int x;
public:
  void foo()          // имя из класса Base скрывается именем из класса Derived
  { x = 0; } } ;

class Derived : public Base {
public:
  void foo()          // имя из класса Derived скрывает имя из класса Base
  { x = 0; } } ;

class Client {
public:
  Client()
  { Derived d;
    d.foo(); } } ;    // вызов функции-члена Derived

int main()
{ Client c;           // создание объекта, вызов конструктора
  return 0; }
```

Обратите внимание, что в данном примере не вводится глобальная область действия. Если ни класс Base, ни класс Derived (и ни один из их предков) не имеет функции-члена foo(), то вызов функции d.foo() даст синтаксическую ошибку. Если бы функция foo() определялась в глобальной области действия, то вызов функции d.foo() в любом случае не вызывал бы эту глобальную функцию.

```
void foo()
{ int x = 0; }
```

Эта глобальная функция не скрывается функцией-членом foo() в классе Derived (или Base), поскольку имеет другой интерфейс. Такая функция-член вызывается с помощью целевого объекта, а глобальная — когда применяется только имя функции.

```
foo();                // вызов глобальной функции
```

Обсуждаемые вызовы функции имеют другую синтаксическую форму:

```
d.foo();              // вызов функции-члена
```

Данная синтаксическая форма не может быть реализована за счет вызова глобальной функции. Она включает в себя целевой объект, следовательно, может быть реализована только функцией-членом класса.

Перегрузка и сокрытие имен

Обратите внимание, что в приведенном выше обсуждении не упоминалась сигнатура функции. Сигнатуры функции не являются здесь значимым фактором. Они не учитываются.

Сигнатура функции не имеет значения, когда компилятор решает, соответствует ли фактический аргумент формальным параметрам функции. Между тем для разрешения имен вложенных областей при наследовании это не важно. Что произойдет, если функция, обнаруженная в производном классе, не подходит с точки

зрения соответствия аргументов? Генерируется синтаксическая ошибка. А что, если функция в базовом классе подходит лучше — имеет то же имя и соответствующую сигнатуру? Слишком поздно. У базового класса нет шансов.

К сожалению, многие программисты понимают это не до конца. Проработайте правила вложенных областей действия и убедитесь, что вы все поняли. Из следующего примера было исключено все лишнее, не относящееся к вопросу сокрытия имен во вложенных областях действия, и оставлен лишь небольшой фрагмент.

Листинг 13.14 показывает упрощенную часть иерархии классов из бухгалтерской программы. Здесь используются только классы Account и CheckingAccount. В производном классе переопределяется функция-член withdraw() базового класса. В клиенте определяются объекты CheckingAccount, им отправляются сообщения, принадлежащие базовому классу (getBal() и deposit()) или производному классу (withdraw()). Результат программы продемонстрирован на рис. 13.10.

```
Итоговые балансы
  объект расчетного счета:      1099.8
```

Рис. 13.10. *Результат программы из листинга 13.14*

Листинг 13.14. Пример иерархии наследования для классов Account

```
#include <iostream>
using namespace std;

class Account {                              // базовый класс
protected:
   double balance;

public:
   Account(double initBalance = 0)
   { balance = initBalance; }

   double getBal()                           // наследование без изменений
   { return balance; }

   void withdraw(double amount)              // переопределяется в производном классе
   { if (balance > amount)
        balance -= amount; }

   void deposit(double amount)               // наследуется без изменений
   { balance += amount; }
} ;

class CheckingAccount : public Account {     // производный класс
   double fee;

public:
   CheckingAccount(double initBalance)
   { balance = initBalance; fee = 0.2; }

   void withdraw(double amount)              // скрывает метод базового класса
   { if (balance > amount)
        balance = balance - amount - fee; }
} ;

int main()
{ CheckingAccount a1(1000);                  // объект производного класса
   a1.withdraw(100);                         // метод производного класса
   a1.deposit(200);                          // метод базового класса
   cout << " Итоговые балансы\n";
   cout << " объект CheckingAccount: " << a1.getBal() << endl;
   return 0;
   }
```

Хотя в этом листинге представлено всего несколько строк, в реальном варианте он занимает 200 страниц. Программа эволюционировала в соответствии с изменениями в условиях бизнеса. Для одного из изменений потребовалось добавить в класс CheckingAccount еще одну функцию deposit(), которую можно было бы использовать для электронных платежей. При таком платеже плата за операцию зависит от источника транзакции и суммы перевода. Она может вычисляться клиентом и передаваться классу CheckingAccount в виде аргумента. Следовательно, простой способ поддержки этого изменения состоял в написании еще одной функции deposit() с двумя параметрами.

```
void CheckingAccount::deposit (double amount, double fee)
  { balance = balance + amount - fee; }
```

Клиентская часть для обработки международных платежей и вычисления платы за операцию потребует добавить к программе только несколько страниц. Пример вызова в клиенте новой функции deposit().

```
a1.deposit(200,5);              // метод производного класса
```

До сих пор все было в порядке. Изменения прошли хорошо, новый программный код работает отлично. Однако в ходе интеграции системы возникает проблема. 200 страниц исходного кода, которые до сих пор превосходно работали, теперь не функционируют должным образом. Программа даже не компилируется.

До C++ мы работали со многими языками программирования, и ранее ничего подобного не происходило. Предположим, что и вы, какие бы языки вам прежде не встречались, не видели ничего похожего. Это еще один вклад C++ в технику разработки ПО.

Все мы оказывались в ситуациях, когда добавление нового исходного кода "выводит из строя" существующий, и прежняя программа перестает корректно работать. Обычно это случается, если новый код влияет на данные, с которыми работает существующая программа. Но ранее написанные операторы всегда компилируются. В традиционных языках при добавлении нового программного кода в существующем коде не возникают синтаксические ошибки.

В C++ программа состоит из классов, связанных друг с другом не только через данные, но и через наследование. Конечно, новый код может сделать существующую программу семантически некорректной из-за неверной работы с данными. Такое случается в любом языке программирования. Однако прежняя программа может стать семантически некорректной и через наследование! Это бывает только в C++. Вот почему мы постоянно говорим об интуиции программиста, необходимости знать правила и умении чувствовать корректный и некорректный исходный код C++.

Определим причины возникновения таких необычных трудностей программирования. Для этого снова рассмотрим класс CheckingAccount.

```
class CheckingAccount : public Account {
  double fee;
public:
  CheckingAccount(double initBalance)
  { balance = initBalance; fee = 0.2; }
  void withdraw(double amount)              // скрывает метод базового класса
  { if (balance > amount )
       balance = balance - amount - fee; }
  void deposit(double amount, double fee)   // новый метод
  { balance = balance + amount - fee; }     // скрывает метод базового класса
} ;
```

Когда компилятор обрабатывает существующие 200 страниц клиента, вызов функции-члена deposit() предполагает обращение к функции базового класса Account::deposit() с одним параметром.

```
a1.deposit(200);                // метод базового класса?
```

Согласно правилам разрешения имен, о которых рассказывалось выше, компилятор анализирует тип получателя сообщения, находит объект a1, принадлежащий классу CheckingAccount, ищет в классе CheckingAccount функцию-член deposit(). Найдя ее, он прекращает поиск по цепочке наследования. Далее следует проверка сигнатуры. Компилятор обнаруживает, что метод CheckingAccount::deposit(), найденный в производном классе, имеет два параметра. Между тем клиент (вызывающий метод базового класса) подставляет только один параметр. В результате выводится сообщение о синтаксической ошибке.

Возможно, замечание о прогулке на танке по окрестностям следовало приберечь для этого случая. Мы не сомневались в корректности программы и полагали, что дело в очередной ошибке компилятора. (Не важно, какой это компилятор. При изучении нового языка вы всегда найдете в компиляторе несколько новых ошибок, пока не освоите язык получше.)

Хотелось бы, чтобы компилятор рассматривал эту ситуацию как перегрузку имен функций. Есть функция deposit() в базовом классе с одним параметром. Имеется функция deposit() с двумя параметрами в производном классе. Однако объект производного класса является также объектом базового класса! Он унаследовал функцию deposit() с одним параметром. В производном классе получилось две функции deposit() — с одним и с двумя параметрами. Было бы неплохо, если бы компилятор использовал правила перегрузки имен функций и выбрал верную функцию — с одним параметром. Между тем, как уже отмечалось выше, когда метод базового класса скрывается методом производного класса, у метода базового класса не остается шансов. Перегрузка применяется к нескольким функциям в одной области действия. "Сокрытие" имеет место для функций в разных вложенных областях действия.



Осторожно! C++ поддерживает перегрузку имен функций только в одной области действия. В независимых областях действия конфликт имен не возникает, поэтому можно использовать одно и то же имя функции с одной или с разными сигнатурами. Во вложенных областях действия имя во вложенной области скрывает имя во внешней, независимо от того, одинаковые у них сигнатуры или разные. Если классы соотносятся через наследование, то имя функции в производном классе скрывает имя функции в базовом классе. Сигнатуры здесь не имеют значения.

На рис. 13.11 показан объект производного класса с этими двумя функциями — одной из базового класса, другой из производного. Вертикальная стрелка от клиента демонстрирует, что компилятор начинает поиск с производного класса.

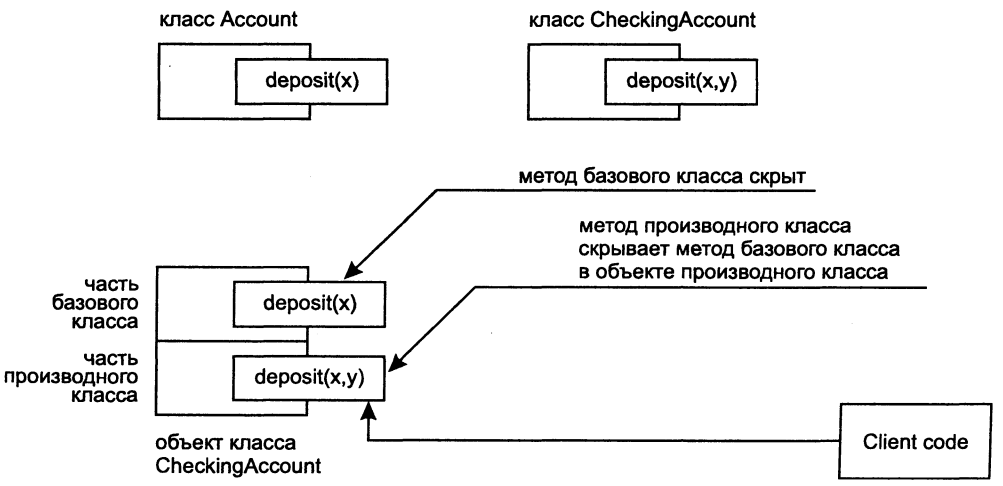


Рис. 13.11. *Как метод производного класса скрывает метод базового класса в объекте производного класса*

Он прекращает поиск, как только находит подходящее имя (независимо от сигнатуры) и не пытается перейти в базовый класс и применить правила перегрузки имен. Если концепция вложенных областей действия при наследовании звучит для вас слишком абстрактно, используйте этот рисунок как напоминание, что поиск прекращается при первом совпадении.

Вызов метода базового класса, скрытого производным классом

Существует несколько способов исправить эту ситуацию. Один из них — указать в клиенте, какую именно функцию нужно вызывать. С данной задачей справится операция области действия.

```
int main()
{ CheckingAccount a1(1000);                    // объект производного класса
  a1.withdraw(100);                            // метод производного класса
//  a1.deposit(200);                           // синтаксическая ошибка
  a1.Account::deposit(200);                    // решение проблемы
  cout << " Итоговые балансы\n";
  cout << " объект CheckingAccount: " << a1.getBal() << endl;
  return 0; }
```

При данном решении не требуется вносить изменения в существующий программный код, и это является его недостатком. Преимущество объектно-ориентированного подхода в том, что он способствует дополнению существующего программного кода, а не его модификации. Между тем приведенное решение очень трудоемко и способствует появлению ошибок.

С точки зрения разработки ПО оно противоречит принципам обсуждавшихся ранее методов разработки программ C++. На кого в этой ситуации возлагается основной объем работ? На клиентскую часть. А кто должен нести это бремя, согласно принципам разработки? Серверная часть приложения. В данном решении не удается перенести обязанности на серверные классы. Нужно обеспечить вызов функции базового класса, указав это явным образом. Используйте метод "грубой силы".

В работе воспользуйтесь критерием переноса обязанностей на серверные классы. Посмотрите на иерархию наследования `Account`. Добавьте к этим классам метод (или методы), благодаря которым проблема бы исчезла. Методы желательно добавить в иерархию наследования, поскольку клиента обслуживают классы, а задача состоит в переносе обязанностей на серверные классы.

Одно из решений заключается в перегрузке метода `deposit()` в базовом, а не в производном классе. Так как обе функции принадлежат одному классу, т. е. находятся в одной области действия, вполне законно использовать перегрузку имен функций C++. Обе функции наследуются производным классом и могут вызываться через объект производного класса, выступающий в роли получателя сообщения. Пример такого решения:

```
class Account {                          // базовый класс
protected:
  double balance;
public:
  Account(double initBalance=0)
  { balance = initBalance;}
  double getBal()                        // наследуется без изменений
  { return balance; }
  void withdraw(double amount)           // переопределяется в производном классе
  { if (balance > amount)
       balance -= amount; }
```

```
  void deposit(double amount)                // наследуется без изменений
  { balance += amount }
  void deposit(double amount, double fee)    // совмещение deposit()
  { balance = balance + amount - fee; } } ;

class CheckingAccount : public Account {     // производный класс
  double fee;
public:
  CheckingAccount(double initBalance)
  { balance = initBalance; fee = 0.2; }
  void withdraw(double amount)               // скрывает метод базового класса
  { if (balance > amount)
       balance = balance - amount - fee; } } ;

int main()
{ CheckingAccount a1(1000);                  // объект производного класса
  a1.withdraw(100);                          // метод производного класса
  a1.deposit(200);                           // существующий код клиента
  a1. deposit(200,5);                        // новый код клиента
  cout << " Итоговые балансы\n";
  cout << " объект CheckingAccount: " << a1.getBal() << endl;
  return 0; }
```

Хорошее решение. Обратите внимание, что оно реализовано в виде добавления программного кода к серверному классу, а не в форме модификации клиента. В этом случае работа переносится на класс Account, и это хорошо. Однако оно требует открывать и изменять базовый, а не производный класс. Это нежелательно из соображений управления конфигурацией. Чем выше класс в иерархии наследования, тем больше нужно защищать его от изменений, так как они могут повлиять на производные классы. И наоборот, чем ниже класс в иерархии наследования, тем безопаснее его открывать и изменять.

Еще одна проблема, связанная с этим решением, состоит в том, что правила области действия позволяют функции-члену базового класса обращаться только к элементам данных того же класса, но не к элементам данных производного класса. В приведенном примере вы не видите проблему, так как обоим методам deposit() нужны только данные базового класса. Часто ситуация бывает иной. Новому методу могут потребоваться данные, определенные в производном классе и недоступные в базовом. Например, стандартная плата за снятие денег со счета может применяться и к операции пополнения счета. Тогда новый метод deposit() можно реализовать только в производном классе.

```
void CkeckingAccount::deposit(double amount, double fee)
{ balance = balance + amount - fee - CheckingAccount::fee; }
```

Однако включение нового метода deposit() в производный класс возвращает нас к проблеме вложенных областей действия. Эта функция скрывает функцию базового класса deposit() с одним аргументом, что синтаксически некорректно.

Лучше всего разместить функцию deposit() в производном классе. Для вызовов функции deposit() с одним допустимым параметром можно совместить функцию deposit() в производном классе, а не в базовом. Кроме того, производный класс — это серверный класс клиента, и такое решение состоит в переносе обязанностей на серверный класс.



Советуем Всегда ищите способы писать программу C++ так, чтобы можно было перенести обязанности с клиента на сервер. В результате клиент будет выражать смысл вычислений, а не их детали. Этот общий принцип сослужит вам хорошую службу.

Данное решение представлено в листинге 13.5. Производный класс содержит две функции deposit() с двумя разными сигнатурами. Так как обе они принадлежат одному классу, действуют правила перегрузки имен. И новый, и существующий программный код будет вызывать функции-члены производного класса, применяя разные сигнатуры. Функция-член с одним аргументом должна просто вызывать функцию-член базового класса с тем же именем (передавая работу серверу). Результат программы показан на рис. 13.12.

```
Итоговые балансы
  объект расчетного счета:      1294.6
```

Рис. 13.12. *Результат программы из листинга 13.15*

Листинг 13.15. Пример иерархии наследования классов Account

```
#include <iostream>
using namespace std;

class Account {                                  // базовый класс
protected:
  double balance;
public:
  Account(double initBalance = 0)
  { balance = initBalance; }
  double getBal()                                // наследуется без изменений
  { return balance; }
  void withdraw(double amount)                   // переопределяется в производном классе
  { if (balance > amount)
       balance -= amount; }
  void deposit(double amount)                    // наследуется без изменений
  { balance += amount; }
} ;

class CheckingAccount : public Account {         // производный класс
  double fee;

public:
  CheckingAccount(double initBalance)
  { balance = initBalance; fee = 0.2; }

  void withdraw(double amount)
  { if (balance > amount)
       balance = balance - amount - fee; }

  void deposit(double amount)                    // скрывает метод базового класса
  { Account::deposit(amount); }                  // вызов базовой функции

  void deposit(double amount, double fee)        // скрывает метод базового класса
  { balance = balance + amount - fee - CheckingAccount::fee; }
} ;

int main()
{
  CheckingAccount a1(1000);                      // скрывает базовый метод
  a1.withdraw(100);                              // метод производного класса
  a1.deposit(200);                               // существующий клиентский код
  a1.deposit(200,5);
  cout << " Итоговые балансы\n";
  cout << "     объект CheckingAccount: " << a1.getBal() << endl;
  return 0;
  }
```

Вас не должно пугать использование в этом примере операций области действия. Функцию deposit() с одним параметром в классе CheckingAccount можно было бы записать следующим образом:

```
void CheckingAccount::deposit(double amount)      // скрывает базовый метод
  { deposit(amount); }                            // бесконечный рекурсивный вызов
```

Когда компилятор обрабатывает тело данной функции, в поисках соответствия он сравнивает имя deposit() и локальное для функции имя. Среди локальных имен такого нет, поэтому компилятор ищет совпадение среди компонентов класса. Он находит имя CheckingAccount::deposit() и генерирует вызов этой функции. В результате он интерпретируется как бесконечный рекурсивный вызов.

Операция области действия в листинге 13.15 вынуждает компилятор генерировать вызов функции Account::deposit() базового класса, чтобы избежать рекурсии. Обратите внимание, что обязанности по организации иерархии классов и включению функции в тот или иной класс переносятся на серверные классы, а не на клиента, как в первом варианте.

Функцию deposit() в классе CheckingAccount с двумя параметрами можно записать так:

```
void CheckingAccount::deposit(double amount, double fee)
  { balance = balance + amount - fee - fee; }
```

Когда компилятор обрабатывает тело этой функции, он ищет соответствие между именем fee и именем, локальным для функции. Это имя второго параметра функции. Хотя класс CheckingAccount содержит элемент данных fee, его скрывает имя параметра функции. Для доступа к элементу данных fee в листинге 13.15 нужно использовать операцию области действия.

Применение наследования для развития программы

Часто хороший способ справиться с подобной эволюцией программы — избежать проблем и методов их устранения. Трудности с международными электронными платежами в листингах 13.14 и 13.15 возникают из-за попытки изменить существующий программный код (классы Account и CheckingAccount), адаптировав его к новым условиям.

С точки зрения традиционного программирования это естественное мышление. Объектно-ориентированное программирование, поддерживаемое C++, предполагает изменение традиционного мышления. Вместо поиска способов изменения существующего программного кода можно рассмотреть способы *наследования* из имеющихся классов для поддержки новых требований.

Здесь нужно уточнить: речь идет о новом способе *мышления* при написании программы. Применение наследования означает разработку нового программного кода вместо модификации существующего. Те, кому приходилось вносить изменения в существующие программы, знают, что это два совершенно разных подхода. C++ предлагает такой способ для решения проблемы с международными платежами: оставить в покое написанные 200 страниц исходного кода, "заморозить" классы Account и CheckingAccount и ввести для поддержки нового клиента еще один производный класс:

```
class InternationalAccount : public CheckingAccount {      // здорово!
public:
  InternationalAccount(double initBalance)
  { balance = initBalance; }
  void deposit(double amount, double fee)          // скрывает базовый метод
  { balance = balance + amount - fee - CheckingAccount::fee; }
} ;
```

Данное решение показано в листинге 13.16. Классы Account и CheckingAccount здесь те же, что в листинге 13.14. В еще одном производном классе InternationalAccount не введены дополнительные элементы данных и введена только одна функция-член deposit(), отвечающая новым требованиям клиента. Поскольку объекты, являющиеся получателями сообщений deposit() с разным числом параметров, принадлежат к разным классам, вопрос сокрытия или перегрузки здесь не возникает. Объект a1 получает сообщение с одним параметром, и компилятор вызывает функцию базового класса. Объект a2 получает сообщение с двумя параметрами, и компилятор вызывает функцию в производном от CheckingAccount классе InternationalAccount. Результат программы представлен на рис. 13.13.

```
Итоговые балансы
   Первый объект CheckingAccount: 1099.8
   Второй объект CheckingAccount: 1194.8
```

Рис. 13.13. *Результат программы из листинга 13.16*

Листинг 13.16. Пример улучшенной иерархии классов Account

```
#include <iostream>
using namespace std;

class Account {                              // базовый класс
protected:
   double balance;
public:
   Account(double initBalance = 0)
   { balance = initBalance; }
   double getBal()                           // наследуется без изменений
   { return balance; }
   void withdraw(double amount)              // переопределяется в производном классе
   { if (balance > amount)
        balance -= amount; }
   void deposit(double amount)               // наследуется без изменений
   { balance += amount; }
} ;

class CheckingAccount : public Account {     // производный класс
protected:
   double fee;

public:
   CheckingAccount(double initBalance = 0)
   { balance = initBalance; fee = 0.2; }

   void withdraw(double amount)              // скрывает метод базового класса
   { if (balance > amount)
        balance = balance - amount - fee; }
} ;

class InternationalAccount : public CheckingAccount {        // здорово!
public:
   InternationalAccount(double initBalance)
   { balance = initBalance; }

   void deposit(double amount, double fee)   // скрывает базовый метод
   { balance = balance + amount - fee - CheckingAccount::fee; }
} ;

int main()
{ CheckingAccount a1(1000);                  // скрывает базовый метод
   a1.withdraw(100);                         // метод производного класса
   a1.deposit(200);                          // метод базового класса
```

```
InternationalAccount a2(1000);              // новый серверный объект
a2.deposit(200,5);                          // метод производного класса
cout << " Итоговые балансы\n";
cout << " Первый объект CheckingAccount: "
     << a1.getBal() << endl;
cout << " Второй объект CheckingAccount: "
     << a2.getBal() << endl;
return 0;
}
```

В этом примере из классов создаются новые производные классы, отвечающие только за новые функции программы. Применение наследования C++ — краеугольный камень данного нового подхода к сопровождению ПО: написание нового программного кода вместо модификации существующего.

На самом деле класс Account нуждается в некоторой модификации. Первый способ состоит в том, чтобы сделать закрытый элемент данных fee защищенным (protected) и дать возможность новому производному классу InternationalAccount обращаться к этим данным. Еще один подход может заключаться в добавлении к классу CheckingAccount функции-члена, получающей значение этого элемента данных. Клиент (InternationalAccount) мог бы использовать эту функцию для доступа к данным базового класса. Как уже говорилось выше, предпочтительнее сделать доступными для одного-двух производных классов несколько элементов данных, чем создавать набор функций, которые можно применять только в этих производных классах.

Еще один способ избежать этой модификации существующего класса CheckingAccount — проявить большую дальновидность на этапе проектирования. Зачем делать элементы данных закрытыми? Согласно принципам объектно-ориентированного программирования, причин здесь несколько:

- Нежелательно создавать в клиенте зависимости от имен элементов данных серверного класса.
- Не хочется усложнять клиента с помощью операций с данными.
- Клиент не должен знать об архитектуре сервера больше, чем необходимо.
- Желательно, чтобы клиент вызывал методы сервера, имена которых поясняют их действия.
- Хотелось бы перенести на серверы детали низкого уровня.

Обратите внимание, что этих целей можно достичь, объявив компоненты серверного класса не private, а protected. Ключевое слово protected работает подобно другим модификаторам прав доступа — private и public — относительно разных категорий пользователей классов. Для производных классов, связанных с классом отношением наследования, ключевое слово protected работает как public. Оно позволяет производным классам непосредственно обращаться к компонентам базового класса. Для классов-клиентов, не связанных с классом наследованием, ключевое слово protected функционирует как private. Если предполагается дальнейшее развитие программы через наследование, используйте доступ protected, а не private.



Советуем Всегда рассматривайте способы использования наследования для развития программы C++. Переносите обязанности с клиентского класса на новые производные классы.

Важным вопросом является развитие программы, а не ее первоначальная разработка. При проектировании программы некоторые ключевые базовые классы могут оказаться на верхнем уровне в иерархии наследования. При большом числе потенциальных пользователей классов приобретают важность проблемы инкапсуляции, сокрытия информации, переноса обязанностей на серверы. Для этих ключевых классов желательно использовать модификаторы `private`. Тем самым вы вынуждаете производные классы применять функции доступа. Для развития программы применяйте при создании производных классов классы в нижней части иерархии наследования (хороший пример — класс `CheckingAccount`). Они имеют небольшое число зависимых производных классов, и вопросы инкапсуляции данных, сокрытия информации и переноса обязанностей на серверы становятся неважными.

Вторая модификация — в конструкторе класса `CheckingAccount`. Чтобы избежать синтаксической ошибки в клиенте при создании объекта `CheckingAccount`, добавлено значение параметра по умолчанию. Аналогичные вопросы для составных классов обсуждались в главе 12. В следующем разделе рассматривается создание производных объектов в C++.

Конструкторы и деструкторы для производных классов

При создании объекта производного класса инициализации требуют базовая и производная части. Базовая часть производного класса и его производная часть создаются в строгой последовательности. Вы должны ее знать, чтобы избежать потенциальных синтаксических ошибок и проблем с производительностью.

Вопросы создания объектов при наследовании очень напоминают вопросы создания объектов при композиции классов. При композиции элементы данных объектов создаются (и вызываются их конструкторы) перед выполнением конструктора составного класса. Если соответствующий конструктор не существует, попытка создания составного объекта может привести к синтаксической ошибке. В противном случае создание составного объекта способно снизить производительность программы.

При наследовании классов базовая часть объекта всегда создается (с вызовом конструктора) перед производной частью (и вызовом конструктора производного класса). Если соответствующий базовый конструктор отсутствует, попытка создать объект производного класса может привести к синтаксической ошибке. При наличии соответствующего конструктора базового класса создание объекта производного класса может снизить производительность программы.

Рассмотрим класс `Point` из программ в листингах 13.6—13.9 и попробуем усовершенствовать их, добавив обобщенный конструктор с двумя параметрами.

```
class Point {                          // базовый класс
  int x, y;
public:
  Point(int xi, int yi)                // общий конструктор
    { x = xi; y = yi; }
  void set (int xi, int yi)
    { x = xi; y = yi; }
  void get (int &xp, int &yp) const
    { xp = x; yp = y; } } ;
```

Данное усовершенствование обеспечивает для клиента большую гибкость при создании объектов `Point`. Теперь у клиента есть возможность во время создания объекта задавать координаты точек. Это лучше, чем задавать неинициализированный объект и позднее инициализировать его с помощью вызова функции-члена `set()`.

Что касается класса VisiblePoint из листинга 13.6, то это изменение в базовом классе не требует никакой настройки. На производный класс оно не влияет.

```
class VisiblePoint : public Point {     // наследование public
  int visible;
public:
  void show()
    {  visible = 1; }
  void hide()
    {  visible = 0; }
  void retrieve(int &xp, int &yp, int &vp) const
    {  get(xp,yp);                      // доступен метод базового класса
       vp = visible; } } ;              // доступны данные производного класса
```

Изменения затрагивают клиента производного класса VisiblePoint. Теперь этот фрагмент содержит синтаксические ошибки.

```
int main()
{ VisiblePoint b; int x, y, z;        // определение объекта производного
                                      // класса: ошибка
  b.set(20,40); b.show();     // функции public базового и производного классов
  b.retrieve(x,y,z);                  // вызов функции производного класса
  cout << " Координаты точки: x= "<< x << " y=" << y << endl;
  cout << " Видимость точки: visible="<< z << endl;
  return 0; }
```

Как и в случае любого объекта, память для элементов данных производного класса (в данном случае — для элемента данных visible) выделяется перед выполнением тела конструктора производного класса. Между тем до распределения памяти для данных, описанных в производном классе, создается базовая часть производного объекта. Выделяется память для элементов данных (x и y класса Point) и вызывается конструктор базового класса.

Никакие параметры не передаются конструктору базового класса, поэтому используется конструктор по умолчанию. Так как базовый класс Point предусматривает конструктор, отличный от применяемого по умолчанию, подставляемый системой конструктор не вызывается.

Следовательно, попытка создания объекта производного класса дает синтаксическую ошибку — вызов несуществующей функции. Обратите внимание, что клиент, теперь содержащий ошибку, превосходно работает в программах из листингов 13.6–13.9.

```
VisiblePoint b;              // нет синтаксической ошибки в предыдущих версиях
```

Добавление конструктора в класс Point делает эту строку синтаксически некорректной. Как уже говорилось, в традиционных языках добавление нового программного кода может нарушить работу существующего, но никогда не сделает его синтаксически некорректным. Такая связь между разными частями программы свойственна именно программированию на C++.

Избавиться от проблемы, конечно, можно. В базовом классе нужно использовать конструктор по умолчанию, подставляемый системой или определяемый программистом. Данный конструктор вызывается после распределения памяти для объектов в базовой части производного класса.

```
class Point {                         // базовый класс
  int x, y;
public:
  Point()
    { x = 0; y = 0; }                 // теперь клиент в порядке
```

```
  Point(int xi, int yi)                    // обобщенный конструктор
    { x = xi; y = yi; }
  void set (int xi, int yi)
    { x = xi; y = yi; }
  void get (int &xp, int &yp) const
    { xp = z; yp = y; } } ;
```

Теперь синтаксическая ошибка исчезла. Однако конструктор `Point` работает напрасно, так как клиент сам устанавливает объект `VisiblePoint` на нужную точку плоскости и скрывает либо показывает точку.

```
VisiblePoint b;              // нет синтаксической ошибки
b.set(20,40);                // запись поверх базовой части объекта
b.show();                    // устанавливает производную часть объекта
```

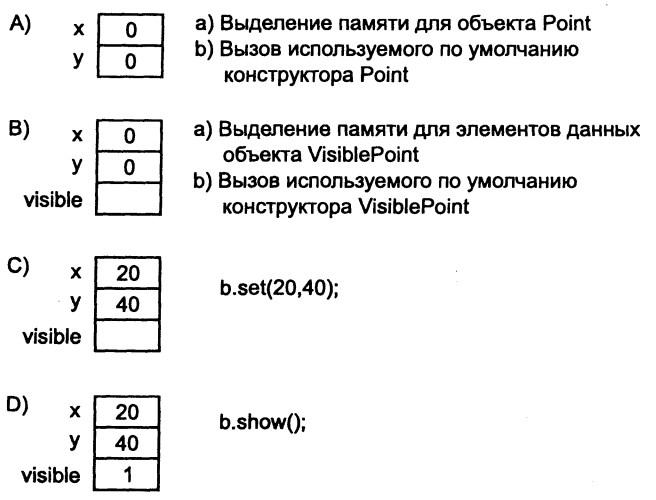


Рис. 13.14. *Шаги распределения памяти при инициализации объекта*

Последовательность событий, связанных с созданием объекта производного класса, представлена на рис. 13.14. Сначала создается базовая часть, затем вызывается используемый по умолчанию конструктор базового класса, задается производный класс, вызывается конструктор производного класса и выполняются последующие операторы в клиенте.

Точка в данном примере создается применяемым по умолчанию конструктором, который инициализирует поля данных сразу после появления базовой части. Затем создается производная часть, и выполняется конструктор производного класса. Если клиенту нужно установить базовую часть в конкретное состояние (а не просто в состояние по умолчанию), то используемый по умолчанию конструктор базового класса работает зря.

Данную архитектуру можно усовершенствовать. Перенесите обязанности по инициализации производного объекта с клиента на конструктор производного класса. В последнем фрагменте названные обязанности выполняются клиентом с помощью передачи сообщений `set()` и `show()` для объекта производного класса. Освободите клиента от этих обязанностей.

Производный класс может получать данные для инициализации своих собственных компонентов и данных базового класса в виде параметров конструктора производного класса. Их можно использовать, чтобы явно установить состояние базовой части в теле конструктора производного класса.

```
class VisiblePoint : public Point {
  int visible;
public:
  VisiblePoint(int xi, int yi, int view)        // параметры для данных
    {  set(xi,yi);
       visible = view; }                        // установка значений полей
                                                // базовой, производной части
       . . . . } ;                        // остальная часть класса VisiblePoint
```

Теперь клиент явно не вызывает функцию `set()` базового класса и свою собственную функцию `show()` или `hide()`. Вместо этого он задает дополнительные параметры при определении объектов производного класса.

```
VisiblePoint b(20,40,1);      // нет необходимости в вызове show() или hide()
```

Один из способов представить вызов функции set() в конструкторе VisiblePoint состоит в том, что компилятор сначала пытается найти локальное соответствие в области действия конструктора, затем — в области действия класса VisiblePoint, а потом в базовом классе Point. Можно сказать также, что вызов функции set() принадлежит базовому классу. Поскольку объект производного класса "является видом" объекта базового класса, для функции set() не требуется целевой объект, так как получателем сообщения является объект производного класса (точнее, базовая часть).

Третий способ представить данный вызов — пожалеть читателя программы и предположить, что писавший ее программист был заинтересован в скорейшем завершении своей работы, а не в том, чтобы сделать программу понятной. При написании программы ее автор знал, к какому классу принадлежит функция set(). Тем не менее читателю предоставлена возможность выбора "способов представления функции", т. е. ему надо догадаться, какая функция имеется в виду. Следовательно, клиент не разрабатывался согласно принципам объектно-ориентированного программирования. В соответствии с данными принципами, клиент (конструктор VisiblePoint) должен быть написан так, чтобы имена в вызовах функций поясняли действия. C++ поддерживает данный подход и допускает применение операции для области действия. Тем самым идеи разработчика программы передаются ее читателям.

```
class VisiblePoint : public Point {
  int visible;
public:
  VisiblePoint(int xi, int yi, int view)          // параметры для данных
    {  Point::set(xi,yi);                         // передача знаний читателям
       visible = view; }
     . . . . } ;                          // остальная часть класса VisiblePoint
```

Мы снова обращаемся к интуиции программиста. В традиционных языках предлагаются некоторые способы для передачи идей разработчика читателям программы, однако C++ сложнее таких языков. В нем одну и ту же программу можно написать многими способами. Следовательно, существует многообразие способов для выражения идей разработчиков, поэтому в C++ намного важнее передавать в исходном коде данные идеи. Здесь вам должна помогать интуиция.



Советуем При разработке программы всегда ищите способы передать свои знания программистам, занимающимся клиентской частью и сопровождением программы. C++ позволяет сделать это посредством самого исходного кода, а не комментариев. C++ часто применяют как традиционный язык, не используя все его возможности.

Использование в конструкторах производных классов списков инициализации

Добавление конструктора к классу VisiblePoint позволяет перенести обязанности с клиента VisiblePoint в код VisiblePoint. Между тем это не устраняет проблемы лишнего вызова конструктора в базовом классе.

Применяемый по умолчанию конструктор базового класса вызывается для базовой части производного объекта. Он выполняется сразу после выделения памяти для базовой части объекта. Так как значения полей базовой части при выполнении тела конструктора производного класса устанавливаются заново, конструктор базового класса вызывается зря.

Если в базовом классе есть отличный от используемого по умолчанию конструктор, то конструктор производного класса может вызывать его вместо применяемого по умолчанию конструктора базового класса. Тем самым устраняется лишний вызов функции.

Обратите внимание, что конструктор базового класса всегда вызывается между распределением памяти для базовой части и вызовом конструктора производного класса. Вопрос лишь в том, какой конструктор вызывается — по умолчанию или отличный от используемого по умолчанию.

Для вызова отличного от используемого по умолчанию конструктора с параметрами C++ поддерживает синтаксис списка инициализации компонентов, который применятся для координации вызовов конструктора в композиции классов.

```
class VisiblePoint : public Point {
  int visible;
public:
  VisiblePoint(int xi, int yi, int view) : Point(xi,yi)          // список
    { visible = view; }                  // нет вызова set()
  . . . . }                              // остальная часть класса VisiblePoint
```

Разница между этими двумя формами списка инициализации очень важна. В композиции классов список инициализации содержит имена объектов-компонентов в виде имен элементов данных класса. При наследовании список инициализации содержит имя производного объекта-компонента в виде имени базового класса.

Основное сходство между этими двумя формами списка инициализации — в вызове конструктора компонента. В композиции классов он вызывается сразу после распределения памяти для элемента данных. В наследовании классов это происходит немедленно после распределения памяти для базовой части производного объекта. Во всех случаях он вызывается перед телом конструктора класса (контейнерного или производного). Если базовая часть производного объекта содержит компоненты других классов или если компоненты составного объекта имеют базовые классы, данная процедура реализуется рекурсивно.

Итак, C++ создает сначала базовую часть производного объекта, затем вызывает конструктор базового класса, потом задает производную часть класса и выполняет тело конструктора производного класса.

Параметры в вызове конструктора, следующие за двоеточием, передаются конструктору базового класса. Они могут быть либо параметрами, передаваемыми из клиента конструктору производного класса (как в последнем примере), либо литеральными значениями, или даже вызовами функций. Ограничений здесь нет.

Если компонент базового класса для инициализации объекта производного класса нуждается в конструкторе по умолчанию, последний он вызывается явно или неявно. Например, нужно инициализировать базовую часть объектов класса `VisiblePoint` "началом координат" экрана. Тогда конструктор `VisiblePoint` можно записать так:

```
class VisiblePoint : public Point {
  int visible;
public:
  VisiblePoint(int view) : Point()     // вызов конструктора по умолчанию
    {  visible = view; }               // нет вызова set()
       . . . . } ;                     // остальная часть класса VisiblePoint
```

С другой стороны, нет необходимости вызывать конструктор базового класса явно. Даже без списка инициализации компилятор автоматически вызывает используемый по умолчанию конструктор базового класса.

```
class VisiblePoint : public Point {
  int visible;
public:
  VisiblePoint(int view)           // неявный вызов конструктора по умолчанию
    { visible = view; }            // нет вызова set()
     . . . . } ;                   // остальная часть класса VisiblePoint
```

При создании объекта производного класса эти две версии конструктора производного класса проходят одинаковые этапы: распределяется память для базовой части объекта, вызывается используемый по умолчанию конструктор базового класса, распределяется память для производной части объекта, затем вызывается конструктор преобразования производного класса.

Можно комбинировать оба списка таким образом, чтобы элементы данных производного класса инициализировались с помощью синтаксиса инициализации компонента.

```
class VisiblePoint : public Point {
  int visible;
public:
  VisiblePoint(int xi, int yi, int view)
     : visible(view), Point(xi,yi)     // что вызывается сначала?
     { }                               // популярная идиома C++
  . . . . } ;                          // остальная часть класса VisiblePoint
```

Вспомним, что элементы данных всегда создаются в порядке их следования в спецификации класса. Для производного класса базовая часть спецификации неявно является первой и предшествует спецификации компонентов производного класса. Несмотря на то, что она выглядит как приведенный выше список инициализации, конструктор базового класса вызывается первым, лишь затем инициализируются элементы данных производного класса. Тело конструктора производного класса всегда выполняется первым.

Классы `Derived` без конструктора встречаются редко. Кроме того, не используется список инициализации.

В данном примере тело конструктора производного класса является пустым. Инициализация всех элементов данных в списке инициализации конструктора не дает никаких преимуществ, но применяется очень часто. По каким-то причинам многие программисты испытывают чувство глубокого удовлетворения, если в теле конструктора производного класса пусто.

Короче говоря, нет необходимости использовать список инициализации в архитектуре конструкторов производного класса, если имеют место следующие два обстоятельства:

1. Базовый класс не содержит конструкторов (при создании базовой части производного объекта вызывается подставляемый системой конструктор, который применяется по умолчанию).
2. Базовый класс включает в себя определяемый программистом конструктор (он вызывается при создании базовой части производного объекта) и конструкторы производного класса (если они имеются), не изменяющие состояния базовой части производного объекта относительно того, что уже сделано конструктором по умолчанию.

Если базовый класс содержит отличный от используемого по умолчанию конструктор, нужно различать две ситуации:

1. Базовый класс не имеет конструктора по умолчанию. Тогда конструкторы производного класса для вызова отличных от применяемых по умолчанию конструкторов базового класса должны использовать синтаксис списков инициализации. Тем самым предотвращаются синтаксические ошибки в определении объекта производного класса.
2. Базовый класс также содержит определяемый программистом конструктор, который вызывается по умолчанию. Конструкторам производного класса не нужно использовать списки инициализации. Сначала вызывается применяемый по умолчанию конструктор `Base`,

затем конструктор производного класса переопределяет произведенные в его теле действия. Лучше с помощью синтаксиса списка инициализации вызвать соответствующий отличный от используемого по умолчанию конструктор базового класса.

Возможно ли создание производного класса без определяемого программистом конструктора? Конечно. Это значит, что ни базовая часть объекта, ни его производная часть не требуют инициализации. Если это происходит, следует еще раз проверить программу. Возможно, в ней что-то упущено.

Деструкторы при наследовании

Ниже приведен плохой пример наследования, но он иллюстрирует вопросы, связанные с применением деструкторов в производных классах.

Здесь требовалось разработать класс Address для хранения фамилий и адресов электронной почты. Поскольку наследование является мощным механизмом организации классов в программе, можно сделать класс Address производным от другого, более простого класса Name, который содержит фамилию человека, имеющего адрес электронной почты. Базовый класс Name включает в себя элемент данных data, ссылающийся на динамически распределяемый массив символов. Конструктор класса динамически распределяет для объекта память и копирует строку-параметр в динамическую память. Деструктор возвращает занятую строкой память в динамически распределяемую область перед уничтожением объекта. Функция get() возвращает указатель на фамилию.

Листинг 13.17. Использование наследования для классов с динамическим управлением памятью

```
#include <iostream>
using namespace std;

class Name {                                    // базовый класс
   char *name;                                  // динамическое управление памятью
protected:
   Name(char nm[]);                             // предотвращает использование объектов
public:
  ~Name();                                      // возвращает динамическую память
   const char* get() const; } ;                 // доступ к содержимому

Name::Name(char nm[])
{ name = new char[strlen(nm)+1];                // выделение динамической памяти
   if (name == NULL { cout << "Нет памяти\n"; exit(1); }
   strcpy(name,nm); }                           // инициализация динамической области памяти

const char* Name::get () const
{ return name; }                                // доступ к закрытым данным

Name::~Name()
   { delete [] name; }                          // возвращает данные объекта

class Address : public Name {                   // производный класс
   char *email;
   Address(const Address&);                     // нет семантики значений
   void operator = (const Address7);
public:
   Address(char name[], char address[]);        // выделение динамической памяти
  ~Address();
   void show() const; } ;                       // вывод данных объекта на экран
```

```
Address::Address(char nm[], char add[]) : Name(nm)
{ email = new char[strlen(addr)+1];
  if (email == NULL) { cout << "Нет памяти\n"; exit(1); }
  strcpy(email,addr); }

Address::~Address()                          // возвращает память объекта
{ delete [] email; }

void Address::show() const                   // выводит на экран данные объекта
{ cout <<" Фамилия: " << Name::get() << endl;
  cout << " Email: " << email << endl; << endl; }

int main()
{
  Address x("Штерн, "shtern@bu.edu");        // код клиента
  x.show();
  return 0;
  }
```

Листинг 13.17 показывает программу, реализующую данную архитектуру. Так как целью примера является демонстрация вопросов динамического управления памятью в базовом и производном классах, программа намеренно упрощена. Именно поэтому здесь не реализуются конструкторы копирования и операции присваивания. Вряд ли потенциальные клиенты должны создавать объекты Name. Класс Name нужен лишь как базовый для класса Address. Во избежание случайных неприятностей здесь предотвращается создание потенциальными клиентами объектов Name — конструктор Name определен как protected. Его нельзя было сделать private, поскольку тогда объекты Address не смогут инициализировать свою базовую часть. Для потенциальных клиентов protected работает как private. Для класса Address во избежание неприятностей конструктор копирования и операция присваивания объявлены private. Результат программы показан на рис. 13.15.

```
Фамилия: Штерн
Email:   shtern@bu.edu
```

Рис. 13.15. *Результат программы из листинга 13.17*

Конструктор класса Name (т. е. конструктор базового класса) выделяет и копирует память для базового класса. Конструктор класса Address (производного) выделяет и копирует память для производного класса.

Кроме того, конструктор Address (перед выполнением его тела) передает значения конструктору Name. Создание экземпляра объекта класса Address предусматривает следующую последовательность действий:

1. Выделяется память для базовой части объекта (указатель name).
2. Вызывается конструктор базового класса, выделяется и инициализируется динамическая область памяти по указателю name.
3. Выделяется память для производной части объекта (указатель email).
4. Вызывается конструктор производного класса, выделяется и инициализируется динамическая область памяти по указателю email.

Порядок вызова деструкторов будет обратным порядку вызова конструкторов. При уничтожении объекта производного класса сначала выполняется деструктор производного класса, затем уничтожаются элементы данных производной части. Вызывается деструктор базового класса и уничтожаются элементы данных в базовой части объекта. Список этих действий будет таков:

1. Вызывается деструктор производного класса и освобождается (возвращается системе) динамическая область памяти по указателю email.

2. Уничтожается производная часть объекта, и его память (указатель email) возвращается системе.
3. Вызывается деструктор базового класса, и динамическая память, на которую ссылается указатель name, возвращается системе.
4. Уничтожается базовая часть объекта, и его память (указатель name) возвращается системе.

Поскольку деструктор класса не имеет параметров, программисту не нужно координировать вызовы деструкторов. Следует лишь убедиться в наличии деструкторов. Отсутствие деструктора приведет к "утечкам памяти".

Базовая часть объекта не должна исчезать первой, так как это сервер производной части объекта. Для компонентов базового класса может потребоваться сохранять целостность компонентов данных производной части объекта.

Можно объединить управление динамической памятью обоих классов в конструкторе и деструкторе класса Address, однако разграничение такого управления по классам будет способствовать модульности программы.

Так как пример очень компактен, взаимосвязь между классами здесь большого значения не имеет. Между тем создание производного класса Address из класса Name подчеркивает существующую взаимосвязь. Адрес — не фамилия, но использование наследования предполагает это. Лучше было бы применить композицию классов.

Итоги

В этой главе вы продолжили исследование связей между классами C++. Связь наследования позволяет применять один класс как основу для другого класса. Таким путем производный класс наследует все элементы данных и функции-члены базового класса. Обычно в производный класс добавляются также дополнительные элементы данных и функции-члены. Иногда в производном классе переопределяются свойства, унаследованные из базового класса.

Применение наследования — хороший способ повышения модульности программы. Вместо проектирования сервера как одного большого фрагмента можно создавать и отлаживать базовые классы, наращивая их функциональность в форме производных классов.

Использование наследования способствует развитию программы. Вместо изменения существующего программного кода можно добавлять новый и затем поддерживать его.

Для применения наследования в C++ вы должны знать множество новых синтаксических деталей. В C++ очень богатая реализация наследования, и часто архитектуру можно осуществить несколькими способами. Это означает, что чрезмерное применение наследования может существенно усложнить программу.

Наследование — хороший инструмент, позволяющий программисту расширить свои возможности и технику разработки ПО. Он может переносить обязанности на серверные классы, передавать в программе идеи разработчика сопровождающему приложение программисту. Это новый способ написания программ.

В данной главе рассматривалась лишь часть того, что следует знать о наследовании в C++. Следующая глава посвящена технике применения наследования.

Глава 14

Выбор между наследованием и композицией

Темы данной главы

- ✔ Выбор метода повторного использования программ
- ✔ Унифицированный язык моделирования
- ✔ Учебный пример: магазин проката
- ✔ Видимость класса и разделение обязанностей
- ✔ Итоги

В данной главе описывается ряд примеров наследования и композиции. Рассмотрим небольшой пример и сравним использование наследования с другими методами программирования.

Попытаемся сравнить использование различных вариантов проектирования для реализации одной и той же программы. В данном случае под "проектированием" подразумевается то же самое, что и в остальной части книги: принятие решения о том, из каких частей (классов) должна состоять программа, и какие обязанности (элементы данных и функции-члены) должны быть назначены каждой части. Сравним различные варианты проектирования и оценим эффективность общих методик, сформулированных в главе 1: передача обязанностей от клиентских классов серверным классам, самодокументированные клиентские программы, написанные в виде вызова серверных методов, и исключение связей между классами. Воспользуемся также специальными критериями низкого уровня: инкапсуляция, сокрытие информации, связность и сцепление.

В данной главе одним из критериев оценки качества проектирования является легкость написания. В этом состоит основной отход от принципов, провозглашенных в главе 1, где подчеркивалась важность легкости чтения программ и доказывалось, что простота при написании обычно достигается за счет легкости чтения, следовательно, этого нужно избегать. В конце концов программа пишется только один раз, когда набирается ее текст, и реальный набор занимает лишь ничтожную часть времени, затраченного на чтение программы при ее отладке, тестировании, повторном использовании или изменении.

Имейте в виду, что наследование является методикой проектирования, направленной на облегчение процесса написания программ. Разработчик серверного класса получает его из базового класса не для того, чтобы улучшить клиентскую

программу. В идеальном случае разработчик клиентской программы не должен заботиться о том, спроектирован ли серверный класс "с чистого листа" или получен из некоторого базового класса (до тех пор, пока серверный класс поддерживает сервисы, которые требуются клиентской программе).

Реальное программирование на C++ отличается от идеального программирования. Использование наследования для облегчения написания серверных классов плохо сочетается с легкостью чтения, как клиентской программы, так и для программиста, осуществляющего сопровождение. В следующей главе будет показано, как использовать наследование и для упрощения клиентской программы.

Для описания связей между классами используется нотация унифицированного языка моделирования (Unified Modeling Language, UML). На сегодняшний день использование UML рассматривается в качестве решающего фактора достижения успеха в проектировании и реализации объектно-ориентированного продукта. Многие организации используют его в своих проектах. UML представляет собой скорее результат политического и технического компромисса, чем шаг вперед в проектировании. Язык был разработан комитетом (Object Management Group.— *Прим. ред.*) с целью унификации некоторых более ранних вариантов нотаций объектно-ориентированного проектирования, которые добавили возможности более подробного описания связей между объектами. Однако каждый член комитета пытался добавить в UML новые возможности. В результате язык располагает избыточными возможностями и очень сложен для изучения.

Жаль. Язык должен быть ненавязчивым. Разработчикам необходима возможность легко выражать свои идеи и понимать без проблем других. Если новичок в данном языке использует неясные или сбивающие с толку операторы, следует предоставить компилятор, который предупреждает об этом разработчика. Ничего подобного в UML нет. Он имеет тенденцию к созданию более сложных диаграмм, чем необходимо. Опыт использования UML показывает, что на его изучение требуется много времени даже при хорошем знании объектно-ориентированного языка. Кроме того, этот язык моделирования огромен и количество возможных вариантов при проектировании настолько велико, что не стоит пытаться исследовать их при изучении объектно-ориентированного языка. Однако базовый вариант UML или любой из его предшественников можно и нужно использовать для описания объектно-ориентированных проектов, реализованных на языке C++.

В этой главе нотация базового UML представлена как инструмент для сравнения наследования и композиции. Нотация UML также используется для представления общих взаимосвязей между объектами программы. Обсуждаются большие примеры, использующие несколько подходов к их проектированию и реализации. Применение UML будет полезным здесь для понимания сложных вопросов проектирования программ.

Выбор методики повторного использования кода

Обсудим относительные преимущества и недостатки использования наследования и композиции. Обе связи являются отношениями "клиент-сервер". Производный класс представляет собой клиента базового класса, а базовый класс — сервер производного класса. Составной класс является клиентом его компонентного класса, а компонентный класс — сервером составного класса. Это означает, что можно наблюдать значительное сходство между программами C++, написанными с использованием различных методик проектирования.

Общим характерным свойством различных проектных решений является разделение работы между клиентскими и серверными классами, независимо от того, выполняется ли это с использованием композиции или любых других методик проектирования. Следовательно, серверный класс должен быть реализован до того, как сможет быть спроектирован клиентский класс. Методики, которые обсуждаются в данной части, могут использоваться как для разработки, так и для развития программ.

Пример связи "клиент-сервер" между классами

В качестве примера связи "клиент-сервер" обсуждается приложение, которое использует класс Circle с элементом данных для радиуса круга таким образом, что клиентская программа может послать сообщения для получения доступа к внутреннему представлению данных объектов Circle.

```
circle c1(2.5), c2(5.0);                    // задание значения радиуса
double len = c1, getLength () ;              // вычисление окружности
double area = c2.getArea(} ;                 // доступ к внутренним данным
c1.set(3.0) ;
double diam = 2 * c1.getRadius[) ;
```

Для поддержки клиентской программы данного вида класс Circle должен реализовать пять общедоступных функций-членов:

- Конструктор с одним параметром целого типа
- Метод getLength(), возвращающий длину окружности
- Метод getRadius(), возвращающий радиус окружности
- Метод set(), изменяющий значение радиуса окружности
- Метод getArea(), возвращающий площадь круга

Обратите внимание, что требуется специальная клиентская программа, определяющая, как предположительно должен выглядеть серверный класс. Этот способ не является единственно возможным способом программирования в C++. При повторном использовании компонентов ПО серверные классы часто проектируются как библиотеки классов. В результате для использования этих общих классов некоторым клиентам приходится приложить больше усилий.

В данной книге подробно не рассматривается проектирование библиотечных классов. Чтобы хорошо их спроектировать, необходимо обеспечить доступ к внутреннему представлению данных и позволить программистам клиентской части манипулировать данными подходящим, с их точки зрения, способом.

Второй способ программирования в C++, поддерживающий связь "клиент-сервер", намного сложнее. В этом случае программист серверной части должен учитывать требования клиента и реализовывать методы, которые отвечают данным требованиям, а не просто передают клиентской программе информацию для обработки.

Также следует отметить, что для доступа к внутреннему представлению данных серверным объектам посылаются сообщения. Какие бы действия не выполнялись методом класса (например, умножение радиуса окружности на два и на PI для вычисления длины окружности), при этом осуществляется доступ к внутреннему представлению данных (в данном случае к радиусу) по запросу клиентской программы, у которой такой доступ отсутствует. Для поддержки требований клиентской части класс Circle должен выглядеть так:

```
class Circle                    // исходный код для повторного использования
{ protected:                    // наследование - одна из опций
    double radius;              // внутренние данные
  public:
    Circle (double r)           // поддержка инициализации
    { radius = r; }
  double getLength() const      // вычисление длины окружности
  { return 2 * PI * radius; }
  double getArea() const        // вычисление площади
  { return PI * radius * radius; }
  double getRadius() const
  { return radius; }
```

```
  void set(double r)
  { radius = r; } } ;          // изменение размера
```

Если вы хотите избежать ошибок, связанных с указанием чисел с плавающей точкой, воспользуйтесь другим вариантом класса Circle.

```
class Circle                    // исходный код для повторного использования
{ protected:                       // одна из опций - наследование
    const double PI;               // должна быть инициализирована в списке
    double radius;                 // внутренние данные
  public:
    Circle (double r) PI (3.1415926536)     // список функции инициализации
    { radius = r; }
    double getLength()             // вычисление длины окружности
    { return 2 * PI * radius; }
    double getArea()               // вычисление площади
    { return PI * radius * radius; }
    double getRadius() const
    { return radius; }
    void set(double r)
    { radius = r; } } ;            // изменение размера
```

Обратите внимание, что указана только одна основная причина использования константы вместо числового литерала — возможность совершения ошибки при наборе одного и того же литерала в различных местах программы. Не обозначена другая распространенная основная причина: удобство изменения значения во время сопровождения. Если не произойдет неожиданное крупное открытие в науке, причин изменения значения PI в ближайшем будущем не видно. Заметьте, что значение PI умножается на два каждый раз при вызове метода getLength(). Это не очень серьезный недостаток, просто подчеркивается, что в данном примере основная цель определения PI как константы состоит в том, чтобы показать еще раз, что список функции инициализации может содержать не только параметры конструктора, но и литеральные аргументы.

Наконец, учтите, что PI определяется как локальное значение для класса Circle. Если это значение требуется другим классам приложения, они должны либо определить его сами, либо получить из класса Circle. Вы же можете объявить элемент данных как общедоступный элемент.

```
class Circle                    // исходный код для повторного использования
{ protected:                    // одна из опций - наследование
    double radius;              // внутренние данные
  public:
    const double PI;            // должна быть инициализирована в списке
  public:
    Circle (double r) : PI (3.1415926536)    // список функции инициализации
    { radius = r; }
  ...} ;                        // остальная часть класса Circle
```

Здесь представлен широко распространенный метод определения общедоступных данных. Если бы PI было определено в том же разделе общедоступных компонентов, что и функция-член класса, то здесь ее можно было бы опустить. Теперь класс Circle определен — но так ли это? При такой структуре класса Circle каждому объекту Circle выделяется отдельная область памяти для значения PI. Тем не менее это значение одно и то же для всех объектов Circle. Выделение памяти для каждого из них излишне. Программисты, работающие с необъектно-ориентированными языками, не сталкиваются с этими вопросами в отличие от программистов C++. Важно развить соответствующую интуицию для определения потенциально избыточных затрат. А для этого надо внимательно и тщательно

исследовать возможности использования образцов (patterns) для каждого элемента данных. Когда элемент данных имеет одно и то же значение для всех объектов класса, в такой ситуации использование статических данных прекрасно отвечает всем требованиям. Ниже приведен класс Circle, в котором для всех его объектов выделяется только одно значение PI:

```
class Circle                          // исходный код для повторного использования
{ protected:                          // наследование является одной из опций
    double radius;                    // внутренние данные
  public:
    static const double PI;           // необходимо инициализировать
  public:
    Circle (double r)                 // список функции инициализации
    { radius = r; }
    double getLength () const         // вычисление окружности
    { return 2 * PI * radius; }
    double getArea () const           // вычисление площади
    { return PI * radius * radius; }
    double getRadius() const
    { return radius; }
    void set(double r)
    { radius = r; } };                // изменение размера

const double Circle::PI = 3.1415926536;
```

Очевидно, что для инициализации статического элемента данных не требуется список инициализации элементов. Он инициализируется в определении, реализованном в том же файле, что и функция-член класса. Подобно функциям-членам, оператор области действия класса определяет, какому из них принадлежит этот элемент данных. Если элемент данных PI требуется определить в другом классе, эти имена не будут конфликтовать друг с другом, поскольку они принадлежат разным классам.

Пример показывает, что программист C++ всегда должен помнить о различных аспектах структуры программы.

Убедитесь, что разнообразие вопросов, о которых всегда следует помнить при написании программ на C++, не слишком суживает область внимания.

Класс Circle определен. Теперь рассмотрим требования класса Cylinder, элементы данных которого описывают радиус и высоту цилиндра.

```
Cylinder cyll(2.5,6.0) , cyl2(5.0,7.5} ;        // инициализация данных
double length =cyll.getLength();                // как в Circle
cyll.set(3.0);
double diam = 2 * cyll.getRadius();             // нет обращения к getArea()
double vol = cyl2.Volume();                     // отсутствует в Circle
```

Классы Circle и Cylinder различаются, но их внутренняя структура одинакова (элемент данных radius) и обеспечивает сервисы, имеющие одинаковые имена и интерфейсы, например getLength(). Именно поэтому вопрос повторного использования класса Circle при проектировании класса Cylinder вполне правомерен.

Пример показывает, что некоторые существующие сервисы Circle не следует делать доступными для объектов Cylinder, например метод getArea(). С другой стороны, клиентам Cylinder могут потребоваться сервисы, недоступные для клиентов Circle, например метод Volume(). Это обычно для большинства контекстов повторного использования. Некоторые из существующих сервисов являются повторно используемыми, некоторые запрещены или игнорируются и добавляются какие-либо новые сервисы.

Предположим, что программа Circle доступна, но класс Cylinder еще не спроектирован. Сходство черт между классами предполагает, что следует попытаться построить класс Cylinder, используя программу класса Circle, для максимального расширения и облегчения повторного использования имеющейся программы.

Для многих такое сходство становится решающим аргументом в пользу наследования. Это слишком упрощенный подход. Наследование используется в программировании очень широко. Рекомендуем выбрать наследование в результате сравнения с другими методами повторного использования. Как выбрать одну методику вместо другой?

Объем и удобство повторного использования кода должны быть главными критериями при выборе методики. Затем надо обратить внимание на объем нового кода, который должен быть написан, и глубину тестирования. Этот пример совсем небольшой, поэтому различия не будут значительными. Однако они покажут, чему следует уделить внимание, принимая решение при выборе способа.

Существуют четыре подхода к повторному использованию кода: повторное использование человеческого интеллекта (т. е. написание кода "с чистого листа"); написание нового класса, чтобы его методы приобретали методы (сервисы) существующего класса; написание нового класса, наследующего существующий класс, чтобы его объекты предоставляли клиентам базовые сервисы; и использование наследования с переопределением некоторых методов. Для данного примера представим перечень основных операций:

1. Человеческий интеллект: напишите новую программу для класса Cylinder "с нуля", используя редактор для копирования в класс Cylinder из программы Circle radius, getLength() и других функций-членов. Добавьте новую программу Cylinder для выполнения задачи, не предусмотренной классом Circle.
2. Приобретение сервисов: используя предположение о том, что каждый цилиндр "включает в себя" круг в качестве объекта, спроектируйте класс Cylinder как составной класс. Объект типа Circle используется как элемент данных класса Cylinder, а методы Cylinder (например, getLength()) посылают сообщения с таким же именем компоненту Circle.
3. Наследование из существующего класса как базового класса: используя предположение о том, что каждый объект-цилиндр является кругом, спроектируйте класс Cylinder как класс, полученный из Circle. При этом не требуется писать программу для методов наследования, например getLength(), поскольку каждый объект Cylinder может ответить на эти сообщения, унаследованные из базового класса Circle.
4. Наследование, но с переопределением некоторых методов: этот подход поддерживает новый способ выполнения существующих операций, например площадь цилиндра должна вычисляться другим способом, исходя из площади круга.

В следующих разделах класс Cylinder реализован с использованием этих методов. Будут показаны преимущества и недостатки каждого подхода.

Повторное использование результатов интеллектуальной деятельности

Повторное использование человеческого интеллекта — обычное явление в программировании, не использующем объектно-ориентированный подход. Кажется, в объектно-ориентированных языках программисты настолько вдохновлены использованием наследования и композиции, что смотрят свысока на старые методы повторного использования программ.

В этом подходе используется прошлый опыт. Программа, написанная ранее, воспроизводится и редактируется в соответствии с новыми требованиями. Предполагается, что недавно был написан и протестирован класс Circle, а теперь требуется написать класс Cylinder. Такой подход называется повторным использованием человеческого интеллекта, потому что повторно используются знания, накопленные при написании подобной программы.

В листинге 14.1 представлена структура класса Cylinder, который использует структуру класса Circle. Здесь воспроизводится часть данных класса Circle (в данном случае элемент данных radius) и добавляется то, что требуется для класса Cylinder (элемент данных height). Воспроизводится конструктор, добавляются параметр и программа инициализации элемента данных height. Копируются методы, которые могут повторно использоваться слово в слово (getLength() и другие). Реализуются методы класса Cylinder, отсутствующие в классе Cylinder (метод Volume()). При этом не обращается внимание на методы класса Circle, которые не требуются в классе Cylinder (метод getArea()). Результаты выполнения программы представлены на рис. 14.1.

```
Окружность первого цилиндра: 15.708
Объем второго цилиндра:      589.049
Диаметр первого цилиндра:    6
```

Рис. 14.1. *Вывод для программы, приведенный из листинга 14.1*

Листинг 14.1. Пример повторного использования программы посредством человеческого интеллекта

```
#include <iostream>
using namespace std;

class Cylinder                                  // новый класс Cylinder
{ protected:
    static const double PI;                     // из класса Circle
    double radius ;                             // из класса Circle
    double height;                              // новая программа

  public:
    Cylinder (double r, double h)               // из класса Circle плюс новая программа
    { radius = r;
      height = h; }                             // новая программа

    double getLength ( ) const
    { return 2 * PI * radius ; }                // из класса Circle

    double getRadius ( ) const                  // из класса Circle
    { return radius; }

    void set (double r)                         // из класса Circle
    { radius = r; }

    double getVolume() const                    // без getArea ()
    { return PI * radius * radius * height ; }  // новая программа
  } ;

const double Cylinder: :PI = 3.1415926536;

int main()
{
  Cylinder cyl1(2.5,6.0), cyl2 (5.0,7.5) ;      // инициализация данных
  double length = cyl1.getLength();             // подобно Circle
  cyl1.set(3.0) ;
  double diam = 2 * cyl1.getRadius();           // отсутствует вызов getArea()
  double vol = cyl2.getVolume() ;               // отсутствует в классе Circle
  cout << " Окружность первого цилиндра: " << length << end1;
```

```
cout << " Объем второго цилиндра: " << vol << endl;
cout << " Диаметр первого цилиндра: " << diam << endl;
return 0;
}
```

Большая часть существующей программы Circle (данные и методы) скопирована дословно. Ненужные методы не включены. Для данных и методов, пропущенных в Circle, должна быть разработана новая программа, которая представлена в классе Cylinder. Ее нужно протестировать. Если существующая программа копируется с помощью текстового редактора, а не набирается с клавиатуры, тестирование программы Circle должно быть минимальным. Поскольку интерфейсы функций Circle не изменились, то тестирование для класса Circle может повторно использоваться и для класса Cylinder.

Применение этого метода способствует повышению производительности программы. Все будут ошеломлены молниеносной скоростью разработки программ. Если ваши коллеги узнают, что метод основывается на предыдущем опыте, они уже не будут так восхищаться. С другой стороны, вас приняли для выполнения этой работы, потому что у вас есть опыт разработки подобных систем.

С точки зрения программной инженерии подобный подход имеет существенные недостатки. Классы Circle и Cylinder связаны друг с другом. У них общие элементы данных и общие функции-члены. Связь между классами Circle и Cylinder существует только в сознании проектировщика Cylinder. Программист, осуществляющий сопровождение, может легко ее пропустить, что приведет к ошибкам.

Повторное использование посредством покупки сервисов

Хорошей практикой считается написание программ C++ таким образом, чтобы объекты, отправляющие сообщения другим объектам в программе, были связаны друг с другом в реальной жизни. Отправка сообщения другому объекту иногда называется *покупкой сервисов* этого объекта.

Обратите внимание, что объекты посылают сообщения другим объектам *программы*, а *не друг другу*. Синтаксически вполне возможно, что объект класса A посылает сообщение объекту класса B, а объект класса B посылает сообщение объекту класса A в одной и той же программе. C++ допускает такое запутанное сотрудничество. Более того, для некоторых программ такая архитектура может быть полезной. Однако в результате это приводит к ненужному усложнению структуры. Именно поэтому в большинстве случаев взаимодействия классов имеется один класс, играющий роль клиентского класса, и существует другой класс, играющий роль серверного класса. Когда метод клиентского класса отправляет сообщение объекту серверного класса, говорится, что один класс "покупает сервисы" другого класса.

Существуют три ситуации, в которых клиентский метод может осуществить доступ к объекту-серверу и отправить ему сообщение:

- Определение объекта-сервера как локальной переменной в клиентском методе.
- Определение объекта-сервера как элемента данных в клиентском классе.
- Получение объекта-сервера как параметра клиентского метода.

Первая ситуация выгодна с точки зрения обмена сообщениями между классами: только одна клиентская функция (где определен объект-сервер) имеет доступ к объекту-серверу. Всегда следует выбирать такой тип связей "клиент-сервер". Часто это невозможно, поскольку доступ к объекту-серверу должен осуществляться с помощью других методов клиентского класса или других классов.

Рассмотрим вторую ситуацию. Объект-сервер доступен всем функциям-членам клиентского класса. Если есть выбор, применяйте тип связи "клиент-сервер", а не используйте объект-сервер как параметр метода клиентского класса.

Третья ситуация наиболее сложная с точки зрения обмена сообщениями между взаимодействующими классами: объект-аргумент используется как сервер функцией, к которой он передается как параметр, и функциями, вызывающими эту серверную функцию. Избегайте этого типа связей. Старайтесь обратиться либо к первой, либо ко второй ситуации (доступ методами только одного клиентского класса).

С точки зрения повторного использования кода именно второй тип связей "клиент-сервер" позволяет создавать серверные классы, которые служат их клиентам, обеспечивая сервисами существующих классов.

В примере связей между классами Circle и Cylinder объект Circle становится членом класса Cylinder. Поскольку делается попытка спроектировать элементы данных как закрытые (или защищенные), сервисы Circle недоступны клиентам Cylinder непосредственно. Для предоставления таких сервисов клиентам (в данном случае метод getLength()), класс Cylinder должен запрашивать элемент данных Circle.

В листинге 14.2 показана эта структура (вывод программы такой же, как и в листинге 14.1). Класс Circle определен явно. Класс Cylinder задает элемент данных класса Circle вместе с дополнительными данными (элемент данных height). Если этот элемент данных был объявлен как public, он доступен клиентской программе Cylinder.

```
Class Cylinder                          // новый класс Cylinder
{ protected:
    double height;                      // новая программа
  public:
    Circle c;                           // без PI, без радиуса
  public:
    Cylinder (double r, double h)       // Из Circle плюс новая программа
       : c(r)                           // список функции инициализации (без PI)
    { height = h; }                     // новая программа
    double getVolume() const            // без getArea()
    { double radius = c.getRadius();              // новая программа
      return Circle::PI * radius * radius * height; }
  } ;                                   // без getLength(), getRadius(), set()
```

Класс Cylinder содержит всего лишь несколько функций-членов. Он не включает методы реализации getLength(), getRadius() и set(), потому что клиентская программа может направить эти сообщения общедоступному элементу данных из класса Cylinder.

```
Cylinder cyl1(2.5,6.0) , cyl2(5.0,7.5);          // инициализация данных
double length = cyl1.c.getLength();    // использование элемента данных Circle
cyl1.c.set(3.0);
double diam = 2 * cyl1.c.getRadius();            // без вызова getArea()
double vol = cyl2.getVolume();                   // отсутствует в Circle
```

Каковы недостатки этой структуры? Данные общедоступны. Клиент использует имя элемента данных класса Cylinder и зависит от структуры класса Cylinder. Чтобы выразить беспокойство о качестве этой структуры, следует упомянуть о разделении обязанностей между классом Cylinder и его клиентской программой. Здесь проектировщику класса Cylinder живется легко. Класс Cylinder только предоставляет метод getVolume() и уклоняется от любых других обязанностей. К какому классу принадлежат методы getLength(), getRadius() и set()? Об этих методах знает клиентская программа, но не ее серверный класс Cylinder. Эти сообщения посылает клиентская программа, а не класс Cylinder.

Цепочный синтаксис для вызова функций в клиентской программе неудобен, однако проблема касается расширения обязанностей проектировщика клиентской программы. Проектировщик (и специалист, отвечающий за сопровождение этой программы) должен изучить сервисы классов Circle и Cylinder. В данном примере определения классов удобно размещены вместе. В реальной жизни их можно разделить. Кроме того, может быть больше двух классов, связанных друг с другом. Также могут отсутствовать какие-либо указания на то, что эти классы (Circle и Cylinder) связаны друг с другом.

Именно поэтому структура в листинге 14.2 — лучший пример покупки сервисов. Элемент данных Circle не является общедоступным в классе Cylinder. (Он защищен, но для клиентской программы недосягаем.) В результате именно класс Cylinder, а не его клиент, знает, к какому классу принадлежат методы getLength(), getRadius() и set(). Класс Cylinder определяет множество однострочников. Единственной задачей этих функций-членов является выполнение двустороннего обмена и отправление сообщения с тем же именем элементу данных из класса Cylinder.

Листинг 14.2. Пример повторного использования программы посредством покупки сервисов элемента данных (композиция класса)

```
include <iostream>
using namespace std ;

class Circle                                    // исходный код для повторного использования
{ protected:                                    // одна из опций - наследование
    double radius;                              // внутренние данные
  public:
    static const double PI                      // требуется инициализировать

  public:
    Circle (double r)                           // конструктор преобразования
    { radius = r ; }

    double getLength() const                    // вычисление длины окружности
    { return 2 * PI * radius; }

    double getArea() const                      // вычисление площади
    { return PI * radius * radius ; }

    double getRadius() const
    { return radius; }

    void set(double r.)
    { radius = r; } };                          // изменение размера

  const double Circle::PI = 3.1415926536;

class Cylinder                                  // новый класс Cylinder
{ protected:
    Circle c;                                   // не указывается PI, не обозначается радиус
    double height;                              // новая программа

  public:
    Cylinder (double r, double h)               // из Circle плюс новая программа
        : c(r)                                  // список функции инициализации (кроме PI)
    { height = h; }                             // новая программа

    double getLength() const
    { return c.getLength(); }                   // из класса Circle

    double getRadius() const                    // из класса Circle
    { return c.getRadius(); }
```

```
    void set(double r)                          // из класса Circle
    { c.set(r); }

    double getVolume() const                    // без getArea()
    { double radius = c.getRadius();            // новая программа
      return Circle::PI * radius * radius * height; }
  } ;

int main()
{  Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5);      // инициализация данных
   double length = cyl1.getLength();            // как в Circle
   cyl1.set(3.0) ;
   double diam = 2 * cyl1.getRadius();          // нет вызова getArea()
   double vol = cyl2.getVolume();               // отсутствует в Circle
   cout << " Circumference of first cylinder: " << length << endl;
   cout << " Volume of the second cylinder: " << vol << endl;
   cout << " Diameter of the first cylinder: " << diam << endl;
   return 0;
    }
```

В этом варианте хорошо поддерживаются как инкапсуляция данных, так и разделение обязанностей. Проектировщик клиентской программы должен знать только сервер Cylinder, а не структуру класса Circle (сервер серверного класса). Связь между классами ясна во время сопровождения.

Метод повторного использования программы может быть быстрее, чем повторная запись "с нуля". Тесты менее требовательны — однострочники легко тестировать. Нет необходимости использовать наследование и связь между базовыми и производными классами.

Проблема может возникнуть из-за необходимости доступа к элементам данных Circle из класса Cylinder. Важно, что класс Circle обеспечивает методы доступа, необходимые классу Cylinder. Некоторые программисты C++ не расположены к быстрому распространению однострочников. Они просты, но слишком надоедливы. Использование наследования исключает эту проблему.

Повторное использование программы с помощью наследования

В настоящее время наиболее популярным методом является повторное использование программы посредством наследования. Большинство базовых сервисов могут наследоваться "как есть". В таких ситуациях этот метод работает хорошо и исключает большинство однострочковых методов, что типично для композиции класса.

В листинге 14.3 показан пример повторного использования программы для класса Circle. Он определяется в качестве базового класса для производного класса Cylinder. Поскольку клиентская программа такая же, как и в листингах 14.1 и 14.2, не удивительно, что вывод для программы соответствует выводу на рис. 14.1.

В предыдущей версии, в листинге 14.2, предполагалось, что цилиндр "имеет" круг. Однако класс Cylinder реализовал метод, общий для обоих классов, отправляющих сообщения элементу данных класса Circle. В этой версии программы предполагается, что цилиндр "является" кругом.

Клиент производного класса Cylinder легко осуществляет доступ к сервисам базового класса. Клиентская программа вызывает их (например, getLength()), как если бы эти сервисы были определены в классе Cylinder. Структура класса Cylinder также несложная. В нем определены только те возможности, которые

отсутствуют в базовом классе Circle. Это так же просто, как и использование композиции с общедоступными элементами данных класса Circle. Несомненно, это легче, чем использование композиции с элементами данных, не объявленными общедоступными в классе Circle (как в листинге 14.2). Для композиции класса класс Cylinder должен реализовывать однострочный метод для каждого метода Circle. Он должен быть доступным для клиентской программы Cylinder. Для наследования подобные однострочники не используются.

Список функции инициализации для конструктора производного класса подобен списку функции инициализации для композиции класса — имя элемента набора данных в листинге 14.2 заменяется именем базового класса в листинге 14.3. Помните, что представляет собой список функции инициализации? Поскольку класс Circle не имеет конструктора, определенного по умолчанию, было бы синтаксически неверно создавать объект класса Cylinder без списка функции инициализации, независимо от того, выполняется ли проектирование с использованием композиции или с использованием наследования.

Итак, проектирование с наследованием либо является настолько же сложным, как проектирование с композицией класса (например, для списков функции инициализации), либо более простым, чем композиция класса. (Однострочные элементы данных не используются.) Это не означает, что проектирование с использованием наследования лучше проектирования с применением композиции.

Листинг 14.3. Пример повторного использования программы посредством наследования

```
#include <iostream>
using namespace std;

class Circle                                    // исходный код для повторного использования
{ protected:                                    // одна из опций - наследование
    double radius;                              // внутренние данные
  public:
    static const double PI;                     // требуется инициализация

  public:
    Circle (double r)                           // конструктор преобразования
    { radius = r; }

    double getLength() const                    // вычисление длины окружности
    { return 2 * PI * radius; }

    double getArea( ) const                     // вычисление площади
    { return PI * radius * radius }

    double getRadius() const
    { return radius; }

    void set(double r)
    { radius = r; } } ;                         // изменение размера

  const double Circle: : PI = 3.1415926536;

class Cylinder : public Circle                  // новый класс Cylinder
{ protected:
    double height;                              // остальные данные в Circle

  public:
     Cylinder (double r, double h)              // из Circle плюс новая программа
         : Circle(r)                            // список функции инициализации (без PI)
     { height = h; }                            // новая программа

    double getVolume() const                    // отсутствует getArea()
    { return height * getArea(); }              // дополнительные возможности
  };
```

```
int main()
{
  Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5) ;      // инициализация данных
  double length = cyl1.getLength () ;           // подобно как в Circle
  cyl1.set (3.0) ;
  double diam = 2 * cyl1.getRadius ();          // отсутствует вызов getArea()
  double vol = cyl2.getVolume () ;              // отсутствует в Circle
  cout << " Circumference of first cylinder: " << length << endl;
  cout << " Volume of the second cylinder: " << vol << endl;
  cout << " Diameter of the first cylinder: " << diam << endl;
  return 0;
   }
```

Главная проблема использования наследования состоит в том, что у разработчика клиентской программы отсутствует один сегмент, который описывает сервисы, предоставляемые серверным классом. В листинге 14.2, где использовалась композиция, этот сегмент программы был спецификацией самого класса Cylinder. В листинге 14.3, где применялось наследование, спецификация класса Cylinder описывает только то, что класс Cylinder добавляет к возможностям базового класса Circle. Остальная часть сервисов, доступных для клиентской программы класса Cylinder, описывается в спецификациях класса Circle. Программист клиентской программы Cylinder обязан изучить сервисы, предоставляемые базовым классом. Ситуация осложняется, если иерархия наследования велика.

Это не проблема для компилятора C++. Компилятор осуществляет поиск по дереву наследования для проверки законности сообщений в клиентской программе. Так же поступают проектировщик и специалист по сопровождению. Но человеку выполнить это намного сложнее, чем компилятору.

Вторая проблема наследования заключается в том, что проектировщик клиентской программы должен достаточно хорошо изучить возможности базового класса и перейти к использованию базовых сервисов, которые производный класс не поддерживает. Например, клиентская программа может вычислять площадь поверхности цилиндра следующим образом:

```
double area = cyl1.getArea( ) ;      // нонсенс - это не площадь поверхности!
```

Вызов приведенной функции выглядит так же, как вызовы методов getLength(), getRadius() и set(). Почему они являются одинаковыми для классов Circle и Cylinder, а метод getArea() отличается от них? Эти вызовы выглядят одинаково для компилятора, для проектировщика клиентской программы Cylinder и для программиста, осуществляющего сопровождение, который может оказаться недостаточно компетентным в сложностях геометрии.

Проектировщик класса Cylinder обеспечивает возможность для клиентской программы Cylinder использовать методы getLength(), getRadius() и set(), но не метод getArea(). Как достичь этого? Один из способов — использование закрытого или защищенного режима наследования.

```
class Cylinder : protected Circle          // новый класс Cylinder
{ protected:
    double height;                         // остальные данные в Circle
  public:
    Cylinder (double r, double h)          // из Circle плюс новая программа
        : Circle(r)                        // список функции инициализации
    { height = h; }                        // новая программа
    double getVolume() const               // отсутствует getArea()
    { return height * getArea(); }         // дополнительные возможности
  } ;
```

Но это слишком много. Клиентская программа Cylinder не может вызвать getArea(), потому что она защищена в классе Cylinder, но методы getLength(), getRadius() и set() также становятся защищенными. Есть два средства. Вы можете явно определить методы getLength(), getRadius() и set() как общедоступные в производном классе Cylinder, но при этом задавать метод getArea() или какие-либо другие методы. Базовые сервисы должны быть закрыты от клиентов производного класса.

```
class Cylinder : protected Circle    // новый класс Cylinder
{ protected:
    double height;                   // остальные данные в Circle
public:
       Circle::getLength;            // getArea() здесь отсутствует
       Circle::getRadius;
       Circle::set;
  public:
    Cylinder (double r, double h)    // из Circle плюс новая программа
        : Circle(r)                  // список функции инициализации (без PI)
    { height = h; }                  // новая программа
    double getVolume() const         // без getArea()
    { return height * getArea(); }   // дополнительные возможности
  } ;
```

Это не так плохо, как кажется. Да, как-то неуклюже создавать общедоступные базовые методы, защищенные в производном классе, а затем снова объявлять их общедоступными. Но с точки зрения программной инженерии это превосходно. Вы получаете явный список общедоступных сервисов, которыми класс Cylinder обеспечивает своих клиентов.

Другое средство иллюстрируется в листинге 14.4. Производный класс Cylinder делает унаследованные методы явно доступными для клиентской программы Cylinder. Исключением являются методы (например, getArea()), которые должны быть недоступны клиентской программе. Это очень похоже на версию с композицией класса в листинге 14.2. Обратите внимание на использование операции явного задания объекта класса внутри этих однострочных функций. В композиции класса это элемент данных, которому направляется сообщение. В наследовании отсутствует явный элемент данных. Вместо него существует базовая часть объекта производного класса. Пропуск операции явного задания объекта приводит к бесконечному рекурсивному вызову.

```
void Cylinder::set(double r)
  { set(r); }                        // неявный рекурсивный вызов
```

Вызов функции set() происходит в области действия класса Cylinder. В соответствии с правилом разрешения вызова функции, компилятор вначале осуществляет поиск имени, которое принадлежит локальной области действия. Поскольку это имя находится в классе Cylinder, компилятор снова вызывает Cylinder::set() и никогда не поднимется по цепочке наследования до вызова метода Circle::set(). Эта реализация эквивалентна следующей функции:

```
void Cylinder::set(double r)
  { Cylinder::set(r); }              // явный рекурсивный вызов
```

Чтобы избежать бесконечной рекурсии, необходимо использовать операцию явного задания объекта Circle.

Листинг 14.4. Пример повторного использования кода посредством защищенного наследования

```
# include <iostream>
using namespace std;

class Circle                                    // исходный код для повторного использования
{ protected:                                    // одна из опций - наследование
    double radius;                              // внутренние данные
  public:
    static const double PI;                     // требуется инициализация

  public:
    Circle (double r)                           // конструктор преобразования
    { radius = r; }

    double getLength() const                    // вычисление длины окружности
    { return 2 * PI * radius; }

    double getArea () const                     // вычисление площади
    { return PI * radius * radius; }

    double getRadius() const
    { return radius; }

    void set(double r)
    { radius = r; } };                          // изменение размера

  const double Circle::PI = 3.1415926536;

class Cylinder : protected Circle               // новый класс Cylinder
{ protected:
    double height;                              // остальные данные в Circle

  public:
    Cylinder (double r, double h)               // из Circle с добавлением новой программы
        : Circle(r)                             // список функции инициализации (без PI)
    { height = h; }                             // новая программа

    double getLength () const
    { return Circle::getLength(); }             // из класса Circle

    double getRadius() const                    // из класса Circle
    { return Circle::getRadius(); }

    void set(double r)                          // из класса Circle
    { Circle::set(r) ; }

    double getVolume() const                    // getArea() отсутствует
    { return height * getArea(); }              // дополнительная возможность
  } ;

int main()
{  Cylinder cyl1(2.5,6.0), cyl2(5.0,7.5);       // инициализация данных
   double length = cyl1.getLength();            // подобно как в Circle
   cyl1.set(3.0);
   double diam = 2 * cyl1.getRadius();          // нет вызова getArea()
   double vol = cyl2.getVolume();               // отсутствует в Circle
   cout << " Circumference of first cylinder: " << length << endl;
   cout << " Volume of the second cylinder: " << vol << endl;
   cout << " Diameter of the first cylinder: " << diam << endl;
   return 0;
}
```

Такое решение исключает оба недостатка использования наследования. Существует явный список сервисов, которые класс Cylinder передает своей клиентской программе. Отсутствует опасность того, что клиентская программа вызовет базовые методы, использование которых является неподходящим для производного класса. С другой стороны, решение с использованием композиции, представленное в листинге 14.2, также не содержит этих двух недостатков. При возможности выбора предпочтительнее использовать композицию вместо защищенного наследования, поскольку концептуально она проще, а связи между классами не настолько сильны, как в защищенном наследовании.

Наследование в повторно определенных функциях

Необходимость подавления некоторых базовых методов в объектах производных классов возникает только в том случае, если наследование используется ненадлежащим образом. В этом случае показано, что объект производного класса не является объектом базового класса. Он владеет этим базовым объектом как элементом данных. По этой причине предпочтительнее использовать композицию вместо защищенного наследования.

Часто связь между классами достаточно близка к связи наследования, и в базовом классе отсутствуют методы, которые должны подавляться. Однако должны быть методы, которые в производном классе интерпретируются иначе. Метод getArea() является хорошим примером. Для объекта базового класса Circle этот метод должен возвращать площадь круга.

```
double Circle::getArea () const              // вычисление площади круга
{ return PI * radius * radius; }
```

Для объекта производного класса Cylinder подобный метод должен возвращать площадь двух кругов, которые содержит цилиндр и площадь боковой поверхности цилиндра.

```
double Cylinder::getArea () const          // вычисление площади Cylinder
{ return 2 * Circle::PI * radius * (radius + height); }
```

Когда метод производного класса скрывает метод базового класса, метод производного класса часто выполняет ту же работу, что и метод базового класса. Программисты C++ любят "документировать" этот факт явно, вызывая метод базового класса из метода производного класса (явно используя операцию для задания объекта базового класса).

```
double Cylinder::getArea () const          // вычисление площади Cylinder
{ double area = Circle::getArea();
  return 2 * (area + Circle::PI * radius * height); }
```

Переопределение базовых методов в производном классе — самая обычная практика программирования. Когда автор программировал на языке COBOL, его шеф рекомендовал использовать разные имена для каждой функции (или фрагмента текста), например COMPUTE-CIRCLE-AREA (вычисление площади круга) и COMPUTE-CYLINDER-AREA (вычисление площади цилиндра). К тому же требовалось использовать тщательно разработанную систему цифровых префиксов, которые указывали бы на то, какому модулю программы принадлежит каждое имя.

В языке C++ эта практика осуждается. Нельзя с уверенностью сказать, вызывается ли требование использовать унифицированные имена (например, getArea()) только правилами хорошего тона. Техническое обоснование подобного подхода состоит в осуществимости использования правил разрешения имен для определения того, какой метод (базовый или производный) должен быть вызван. Как видно из предыдущей главы, эти правила становятся частью интуитивного подхода к программированию.

```
Окружность первого цилиндра: 15.708
Объем второго цилиндра:      589.049
Диаметр первого цилиндра:    6
Площадь первого цилиндра: 169.646
```

Рис. 14.2. *Вывод для программы, представленной в листинге 14.5*

Наследование с переопределением обычно является общедоступным. Часть проектов используется повторно (например, `radius` и `getLength()`), добавляются новые компоненты (например, `height` и `getVolume()`), а некоторые методы переопределяются (например, `getArea()`). Этот вариант программы представлен в листинге 14.5. Клиентская программа `Cylinder` слегка изменена, чтобы продемонстрировать использование метода `getArea()` клиентской программой. Вывод программы показан на рис. 14.2.

Листинг 14.5. Пример повторного использования кода посредством общедоступного наследования с переопределением метода

```
#include <iostream>
using namespace std;

class Circle                                  // исходный код для повторного использования
{ protected:                                  // наследование является одной из опций
    double radius;                            // внутренние данные
  public:
    static const double PI;                   // необходимо инициализировать

  public:
    Circle (double r)                         // конструктор преобразования
    { radius = r; }

    double getLength () const                 // вычисление окружности
    { return 2 * PI * radius; }

    double getArea() const                    // вычисление площади
    { return PI * radius * radius; }

    double getRadius() const
    { return radius; }

    void set(double r)
    { radius = r; } };                        // изменение размера

  const double Circle::PI = 3.1415926536;

class Cylinder : public Circle                // Действительно ли Cylinder является Circle?
  { protected:
    double height;                            // остальные данные определены в Circle

  public:
    Cylinder (double r, double h)             // из Circle плюс новая программа
          : Circle(r)                         // список функции инициализации (без PI)
    { height = h; }                           // новая программа

    double getArea() const                    // вычисление площади Cylinder
    { return 2 * Circle::PI * radius * (radius + height); }

    double getVolume() const
    { return height * getArea (); }           // дополнительная возможность
  } ;

int main()
{
  Cylinder cyl1 (2.5,6.0), cyl2 (5.0,7.5);    // инициализация данных
  double length = cyl1.getLength();           // подобно Circle
  cyl1.set(3.0);
  double diam = 2 * cyl1.getRadius();
```

```
double vol = cyl2.getVolume();                    // отсутствует в Circle
cout << " Circumference of first cylinder: " << length << endl;
cout << " Volume of the second cylinder: " << vol << endl;
cout << " Diameter of the first cylinder: " << diam << endl;
cout << " Area of first cylinder: " << cyl1.getArea()<< endl;
return 0;
}
```

Когда производный класс переопределяет метод базового класса, он использует то же имя метода. В данном примере как у базового, так и у производного методов интерфейс одинаковый. Это часто происходит потому, что обе функции выполняют подобные операции. Объекты операции в некоторой степени отличаются, но их семантика одна и та же. Их интерфейсы также будут одинаковыми.

В правилах C++ об этом ничего не говорится. Вы можете переписать базовую функцию с тем же интерфейсом, но делать это не обязательно. По какой-то причине многие программисты C++ верят, что интерфейс должен быть тот же. Это не так. Изменяет ли проектировщик производного класса интерфейс метода или сохраняет тот, что и в базовом классе, наименование базового метода скрывается от клиентской программы производного класса. Это метод производного класса, который вызывается из клиентской программы. Клиентская программа может использовать базовый метод, но в этом случае потребуется явное задание от объекта базового класса выдать команду компилятору и визуальную подсказку для читателя.

```
Cylinder cyl1(2.5,6.0), cyl2(5.0, 7.5);              // Инициализация данных
  double length = cyl1.getLength() ;                 // подобно Circle
  cyl1.set(3.0) ;
  double diam = 2 * cyl1.getRadius();
  double vol = cyl2.getVolume();                     // отсутствует в Circle
  cout << " Circumference of first cylinder : " << length << endl;
  cout << " Volume of the second cylinder : " << vol << endl;
  cout << " Diameter of the first cylinder: " << diam << endl;
  cout << " Side area of first cylinder: "
       << cyl1.Circle::getArea() << endl;            // визуальная подсказка
```

Достоинства и недостатки наследования и композиции

Наследование является хорошим абстрактным средством. Оно явно подчеркивает концептуальные связи между классами, если они существуют. Например, общность классов Circle и Cylinder лучше всего отражается в структуре программы получением производного класса Cylinder из класса Circle. Эти связи наследования показаны в программе Cylinder. Программисты клиентской части и лица, осуществляющие сопровождение, не должны отдельно изучать классы, сравнивать тексты программы.

За счет использования наследования сокращается объем работ по разработке программы. Существуют и другие способы. Многие программисты все еще верят, что легче разработать сложный класс, чем единый модуль. Например, разработка в первую очередь класса Circle позволяет проектировщику сконцентрироваться на относительно простых вещах (вычисление длины окружности) и попытаться позже найти решение более сложных задач (вычисление объема или площади поверхности цилиндра), когда класс Circle уже разработан.

Однако использование наследования вводит дополнительные неявные зависимости между классами. Они не являются очевидными для проектировщиков или лиц, осуществляющих сопровождение клиентской программы. При изучении

описания производного класса читатель не видит список сервисов, доступных клиентской программе. Читатель должен также изучить базовый класс.

Использование композиции класса составляет хорошую конкуренцию наследованию. Составной класс предоставляет читателю полный список сервисов, которые поддерживаются классом. Композиция также вводит зависимости между классами. Однако эти явные зависимости имеют вид однострочных методов, которые передают работу от объемлющего класса к методам составного класса.

Выбор наследования или композиции зависит от того, насколько похожи связанные классы. Если общих методов немного, а число дополнительных сервисов, которые должны поддерживаться, очень велико, то композиция — подходящий способ. Составной класс будет содержать только несколько однострочных функций, а затраты на их написание будут возмещены за счет использования явного списка поддерживаемых сервисов.

Если количество общих методов относительно велико, а число дополнительных сервисов небольшое, то более подходящим способом является наследование. Многие программисты раздражены необходимостью записывать "бессмысленные" однострочные методы — методы базового класса будут непосредственно наследоваться производным классом. Переопределение базовых методов в производном классе привлекает эстетически и открывает способ для использования полиморфизма (см. следующую главу).

При выборе наследования используйте его наиболее простым способом, например общедоступным выводом. Избегайте защищенного и закрытого наследования.

Часто наследование используется для ускорения работы, без ясной концептуальной связи "is a" между классами. Если есть сомнения, что между классами существует естественная связь "является", следует использовать не наследование, а композицию.

Унифицированный язык моделирования

Традиционные программы пишутся как системы взаимодействующих функций. Объектно-ориентированные программы создаются как системы взаимодействующих классов, которые включают и данные, и функции. В первой части книги основное внимание уделялось методам написания функций. Навыки написания программ обработки данных и реализации связей между функциями являются критическими для создания высококачественных программ на C++.

Во второй части книги рассматривается написание классов. Навыки связывания в одно целое данных и операций и реализация классов как серверов и клиентов являются решающими для создания высококачественных программ на C++.

В этой части книги внимание сосредоточено на написании связанных между собой классов. Здесь рассказывается о том, что навыки реализации связей между классами, например наследованием и композицией, весьма существенны при создании высококачественных программ на C++.

В программе на C++ трудно визуально представить связи между классами, реализованными программой. Более того, когда программа достаточно усложняется, становится трудно визуально представить связи между классами, которые предполагается реализовать.

Цели использования UML

Обычный подход к решению проблемы заключается в использовании графического представления для описания связей объектов реальной жизни — кругов и цилиндров, клиентов и счетов, товарно-материальных ценностей и поставщиков, поведение которых должно имитировать приложение. В этом состоит задача объектно-ориентированного анализа, при котором действия системы описываются в виде взаимодействия классов, а не операций (функций).

Вам надо знать, какие объекты реального мира должны представляться в программе C++ как классы и какие связи между ними должны быть реализованы в программе C++ как связи между классами. В этом состоит задача объектно-ориентированного проектирования, в котором для описания классов и их связей в программе также используется графическое представление.

В отличие от объектно-ориентированного анализа систем, который сосредоточен на описании внешних интерфейсов программы (с пользователями и другими системами), в объектно-ориентированном проектировании основное внимание уделяется описанию структурных компонентов системы: системной архитектуры, размещению подсистем по различным аппаратным компонентам и связям между различными классами. Большинство связей — это связи между объектами, установленными на этапе объектно-ориентированного анализа. Однако в программе могут быть добавлены некоторые дополнительные связи между классами для повышения производительности системы и улучшения интерфейса пользователя.

В двух первых частях книги обсуждались методы создания системных компонентов, которые в меньшей степени зависят друг от друга. Это логичный подход, поскольку зависимости между компонентами системы требуют взаимодействия разработчиков, порождающего лишние ошибки. Но реально не стоит ожидать, что компоненты системы будут полностью независимы друг от друга. Будучи частью одной системы, они должны взаимодействовать с другими компонентами. Важно точно описать такое взаимодействие.

Рассмотрим нотацию UML. В традиционном процессе разработки систем на каждом этапе анализа, проектирования и реализации используются различные методики графического представления для поддержки разнообразных интересов аналитика, проектировщика и программиста. В процессе разработки с использованием объектно-ориентированного подхода аналитик, проектировщик и программист используют одну нотацию. Во-первых, все совместно работающие разработчики — аналитики, проектировщики и программисты — используют одинаковую систему графического представления, следовательно, остается меньше возможностей для неверного понимания или различной интерпретации предположений, не выраженных явно. Во-вторых, отсутствуют радикальные изменения представления на разных этапах процесса разработки, следовательно, возникает меньше ошибок во время преобразования.

UML является мощным языком моделирования. Он позволяет разработчику использовать графическую нотацию для представления связей между взаимодействующими компонентами системы. "Взаимодействие" означает, что компоненты системы знают друг о друге и зависят друг от друга. Графическая нотация основывается на концепции объекта как компонента системы с его элементами данных, функциями-членами и связями. Диаграммы объектов можно обсудить с пользователями и проектировщиками системы, чтобы проверить, правильно ли отражены связи. Позднее эти диаграммы преобразуются в объектно-ориентированные приложения.

Суть объектно-ориентированного программирования заключается в связывании данных и операций. Именно комбинация конкретных элементов данных и операций над ними характеризует класс в C++. В этом определении ничего не говорится о связях. При этом создается неверное впечатление о том, что объединение данных и поведения достаточно хорошо описывает объект.

Объектно-ориентированный анализ и объектно-ориентированное проектирование используют другой подход. Объекты описываются в них как комбинация данных, поведения и связей с другими программными компонентами. Как можно видеть, описание данных и поведения в UML играют скорее базовую роль. Нотация UML в большей степени концентрируется на связях классов и объектов.

В объектно-ориентированном программировании преуменьшается значение связей или ассоциаций, поскольку программисты пытаются добиться как можно большей независимости программных компонентов друг от друга. В результате

все объектно-ориентированные языки предоставляют программисту специальные средства для описания элементов данных и функций-членов. Языки не дают программистам собственные средства для описания взаимосвязей программных компонентов.

В реальной жизни программные компоненты всегда связаны друг с другом. Следовательно, программисты описывают эти связи. Они используют следующие методики: элементы данных с типами, определенными пользователем, или элементы данных, которые являются указателями или ссылками на другие объекты.

Основная задача любой нотации при проектировании, включая UML, состоит в том, чтобы помочь разработчикам описать связи объектов. Когда программа пишется на C++, именно программист принимает решение о связи объектов друг с другом. Если возможно реализовать программу таким образом, чтобы она могла выполнять то, что она должна делать, замечательно. Используя нотацию UML, разработчики могут сравнить различные проектные решения и выбрать такие взаимосвязи, которые (1) достаточны для выполнения задания, и (2) упрощают связи между программными объектами.

В UML объединены три различные нотации для построения графических моделей компьютерных систем. Эти модели помогают разработчикам проанализировать требования к системе. Они обычно описывают функциональные возможности системы, интерфейс пользователя, интерфейсы с другими системами, производительность и надежность. Графические методы пытаются представить эти требования в виде связей между системными компонентами.

Одна система нотации была разработана Гради Бучем (Grady Booch) и его компанией Rational Software Corporation. Она включала несколько представлений системы с отдельными диаграммами для каждого представления. Обозначения объектов на диаграммах Буча имели неправильную форму "облака" и их трудно было нарисовать от руки. Буч был одним из первых, кто понял, что графическое моделирование требует поддержки компьютерных инструментальных средств. Инструментальное средство Rational Rose, разработанное его компанией, является одним из наиболее успешных средств объектно-ориентированного моделирования. С созданием UML Rational Rose была модифицирована для поддержки нотации UML.

Другая нотация была разработана Джеймсом Рамбо (James Rumbaugh) и его коллегами в General Electric. Она получила название Object Modeling Technique (OMT). В дополнение к объектной модели, которая описывает связи между объектами в системе, нотация OMT также включала две другие модели: динамическую и функциональную. Хотя эти две модели были не совсем объектно-ориентированными, они представляли адаптацию двух, хорошо известных методик анализа и проектирования: диаграмм переходов состояний и диаграмм потоков данных. Этот синтез сгладил переход к объектно-ориентированному подходу для разработчиков, которые имели опыт использования этих двух графических нотаций. Вероятно, это основная причина того, что OMT стала новым стандартным подходом.

Третья нотация разработана в Швеции Иваром Якобсоном (Ivar Jacobson) и его компанией Objective Systems. Он представил на рынке свою нотацию под названиями Object-Oriented Software Engineering (OOSE) и Objectory. Эта нотация включает так называемые варианты использования ("use cases"), описывающие взаимодействия между системой и такими внешними действующими лицами, как системный оператор или другие системы.

Каждая нотация содержала рекомендации по ее использованию, во-первых, для объектно-ориентированного анализа, во-вторых, для объектно-ориентированного проектирования и, в-третьих, для объектно-ориентированного программирования. В каждой книге мы пытаемся объяснить, почему объектно-ориентированный подход лучше, чем традиционный. Будем откровенны, эти пояснения не совсем понятны. Они хороши для людей, которые верят в преимущества объектно-ориентированного подхода.

Однако недостатки традиционного подхода к разработке систем были настолько серьезными, что отрасли требовалось нечто, обещающее улучшение по сравнению с существующим состоянием дел. Какой подход выбрать? В дополнение к нотациям, разработанным Бучем, Рамбо и Якобсоном, существовали нотации, описанные Шлеер (Shlaer) и Меллором (Mellor), Йордоном (Yourdon) и Коадом (Coad), Коулменом (Coleman) и др. Фактически это очень длинный список. Все нотации похожи и все они являются вариациями, или расширениями, диаграмм "сущность-связь" для проектирования баз данных, созданных П. Ченом (P. Chen) в 1976 г.

Многие годы авторы различных нотаций обсуждали, какие системы нотации лучше и почему. Основная идея их дискуссий состояла в том, что выбор метода — это очень важное решение. Некоторые эксперты верили, что какой-то подход может быть лучше других для определенного вида систем ПО (например, для систем реального времени), тогда как другой подход может быть лучше для другого вида систем ПО (например, бизнес-приложений). Другие эксперты отстаивали преимущества одного метода перед всеми другими. Эти дискуссии называли "войной методов", хотя различия между конкурирующими подходами были скорее в нотациях, чем в методологии, они были незначительными.

В 1995 г. Буч, Рамбо и Якобсон, "трое друзей", как их назвали, решили создать унифицированную нотацию, которая стала бы доминирующим языком моделирования. UML является результатом их сотрудничества. Они написали книги, описывающие язык UML и способы его использования. Инструментальное средство Rational Rose обеспечивает полную поддержку нотации UML.

Недостаток состоит в том, что нотация UML объединяет различные идеи и, следовательно, сложна. Описывающие ее книги объемны. Однако, когда проектировщик принимает неверное решение в отношении моделирования, нет такого компилятора, который указал бы на ошибку (в отличие от компилятора C++, помогающего выявлять сделанные ошибки). Именно поэтому процесс овладения языком UML более медленный, чем, например, процесс изучения C++.

Хорошая новость заключается в том, что вам совсем не требуется быть специалистом в UML. В этой книге описываются основы UML, которых достаточно для обсуждения связей объектов в программах C++.

Основы UML: нотация обозначений для классов

Объекты UML рассматриваются как экземпляры классов, а классы являются описаниями типов объектов. Класс описывает атрибуты и поведение объекта одного типа. Главным источником классов, которые требуется включить в свою модель UML, является анализ понятий и сущностей области приложения. Для бизнес-приложения классами в модели будут, например Клиент, Предмет, Поставка, Требование, Счет и т. д. Для системы реального времени классы будут включать Датчик, Дисплей, Карточку, Клиента, Кнопку, Двигатель и Замок.

Классы, включенные в модель, помещаются на диаграмму классов. Класс представляется как прямоугольник, разделенный на три части. Верхняя часть содержит имя класса, в средней части расположены атрибуты класса, а в нижней приводится список операций. При реализации класса на языке C++ атрибуты становятся

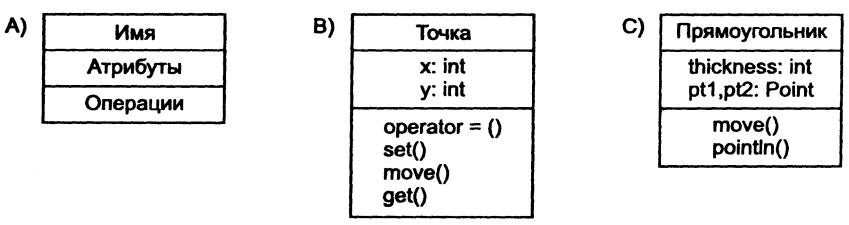


Рис. 14.3. *Примеры на языке UML общего шаблона класса и двух конкретных классов*

элементами данных или полями, а операции — функциями-членами или методами. На рис. 14.3(a) представлен общий вид класса в UML. На рис. 14.3(b) показан пример класса `Point` с атрибутами `x` и `y`, операциями `set()`, `get()` и `move()` и оператором присваивания. На рис. 14.3(c) представлен пример класса `Rectangle` с атрибутами толщины `pt1` и `pt2` и операциями `move()` и `pointIn()`.

Можно заметить, что UML позволяет указать тип атрибутов либо как примитивный (встроенный) тип, либо как библиотечный класс (например, строка), либо как один из классов, определенных в приложении (например, `Point`). Язык UML разрешает определить намного больше, чем просто имя и тип атрибута. Можно указать, является ли атрибут статическим (атрибут области действия — класс), множество разрешенных значений (если атрибут является перечислимым типом), начальное значение атрибута (если имеется), или даже видимость атрибута (общедоступную, закрытую или защищенную). Это необязательно, поскольку часто, особенно в начале процесса анализа и проектирования, разработчики могут не знать точно типы и другие свойства атрибута. Они уточняются позже, во время итеративного процесса проектирования или даже на этапе программирования.

Для операций UML можно определить сигнатуру операции: имя, возвращаемый тип, а также имена и тип параметров. При этом можно задать значения параметров по умолчанию (если требуется), видимость операции (общедоступная, закрытая или защищенная), а также показать, является ли операция статической (операция области действия — класс).

Описание класса в UML может содержать столько деталей, сколько позволяют спецификации класса, записанные на C++. Уделим внимание конкретным деталям нотации UML для атрибутов и операций. Более того, чтобы упростить работу с диаграммами классов, разработчики не выбирают операции класса и обсуждают связи, используя в классах только два раздела — для имени класса и для атрибутов. Для более сложных диаграмм (а большинство диаграмм классов являются сложными) можно опустить часть с атрибутами и представить класс прямоугольником только с именем класса. Это наиболее удобный способ для обсуждения связей классов.

Основы UML: нотация для связей

Сущности реального мира в предметной области могут быть связаны друг с другом. Эти связи представляются на диаграмме классов. Для обозначения связи между классами используется термин "ассоциация". Такая связь означает, что один объект знает о существовании другого объекта, соединен с другим объектом, использует другой объект для достижения своих целей или для каждого объекта одного класса имеется объект другого класса. Это очень важное определение. Ассоциации, будучи реализованными, используются для доступа к некоторому объекту через другие объекты в программе C++.

На рис. 14.4 представлены примеры ассоциаций. На рис. 14.4(a) показано, что каждый объект `Circle` ассоциируется с объектом `Cylinder`, но характер ассоциации

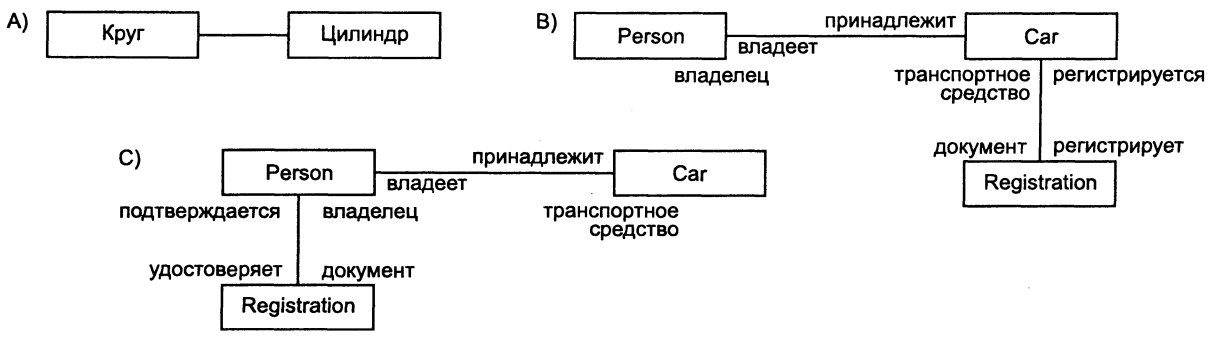


Рис. 14.4. *Примеры ассоциаций классов UML*

не определен. Обратите внимание на то, что в прямоугольнике указано только имя класса, а атрибуты и операции отсутствуют. Их можно было бы указать здесь, но это только усложнило бы диаграмму классов. Это стоит делать только в том случае, если список атрибутов и операций в некоторой степени проясняет характер связей между объектами.

Ассоциации между объектами обычно являются двунаправленными: если объект `Circle` связан с объектом `Cylinder`, то объект `Cylinder` связан с объектом `Circle`. Нотация UML позволяет выразить дополнительную информацию о связях между объектами, указывая имена ассоциаций и назначая роли объектам, ассоциированным друг с другом.

На рис. 14.4(b) показано, что объект `Person` может быть связан с объектом `Car`, а объект `Car` с объектом `Person` и объектом `Registration`. У каждой ассоциации есть две метки, одна для прохождения ассоциации в одном направлении, другая для прохождения ее в противоположном направлении. Чтобы избежать путаницы в отношении направления, в котором текст метки соединяет объекты, можно поместить рядом с ним небольшие стрелочки.

На рис. 14.4(a) не удалось найти подходящее название для ассоциации между объектами `Circle` и `Cylinder`. Они связаны между собой. На рис. 14.4(b) показано, что объект `Person` владеет объектом `Car`, а объект `Car` принадлежит объекту `Person`. К тому же демонстрируется, что объект `Car` регистрируется объектом `Registration`, а объект `Registration` регистрирует объект `Car`. Определяется роль каждого объекта в связи. Объект `Person` играет роль владельца, объект `Car` — транспортного средства, а объект `Registration` — документа.

Однако знание только связей между объектами и ролей, которые они играют, не слишком серьезно помогает понять взаимодействие объектов в реальной жизни и объектов программы во время ее выполнения. Правда, когда разработчик знает совсем немного об области приложения, имена ассоциаций и роли объектов могут быть полезны для проведения анализа в соответствующем направлении.

Часто сравнение различных имен связей в модели классов представляется подобным описанию теории относительности простыми терминами. Основная проблема описания ассоциаций состоит в том, что любое решение является относительным. На рис. 14.4(c) ассоциации между классами `Person`, `Car` и `Registration` описываются с помощью различных связей. Другая возможность состоит в ассоциации каждого класса с двумя другими классами. Какой вариант лучше и почему? На этот вопрос нет точного ответа.

В C++ ассоциации можно реализовать ссылками, которые указывают из одного объекта на другой ассоциированный объект. Другой популярной методикой реализации ассоциации в языке C++ является использование идентификатора объекта как атрибута другого класса. Например, класс `Person` может иметь атрибут, который идентифицирует объект `Car`, ассоциированный с экземпляром `Person`.

Основы UML: нотация для агрегации и обобщения

Агрегация — это особый случаем ассоциации. Она указывает, что два класса соединяются через ассоциацию, но ассоциация является специальной. Эта ассоциация обозначает, что связь имеет смысл как "целое-часть". (Один объект является частью другого объекта либо другой объект содержит первый объект или состоит из нескольких объектов.)

Нотация, используемая в UML для агрегации, такая же, как и для ассоциации — это связь между классами. Для указания агрегированного объекта на конце линии между связью и агрегированным объектом изображается незакрашенный ромб. Ромб можно присоединять только с одной стороны, а не с обеих.

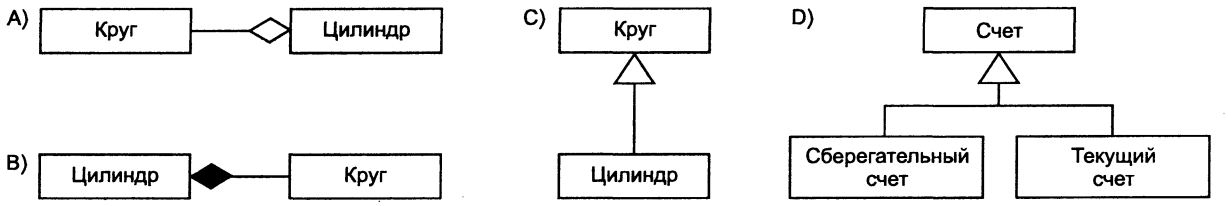


Рис. 14.5. *Примеры совместно используемой агрегации, композиции и наследования*

На рис. 14.5(a) показано, что объект `Circle` является частью объекта `Cylinder`. Фактически не закрашенный ромб указывает, что агрегация используется совместно, и часть может присутствовать одновременно более чем в одной агрегации. В композиции совместное использование не допускается. Нотация UML для композиции такая же, как для совместно используемой агрегации. Однако ромб, присоединенный к агрегации, является сплошной фигурой, а не пустой. На рис. 14.5(b) представлена нотация для композиции, в которой объект `Circle` является частью объекта `Cylinder`, но не может быть частью любого другого объекта.

Поскольку агрегация является особым случаем ассоциации, всегда можно представить связь между объектами как ассоциацию, а не агрегацию. Обратитесь к рис. 14.4(a), где ассоциация используется для моделирования связи объектов `Circle` и `Cylinder`. Однако эта модель менее точная. Задача проектировщика состоит в представлении агрегирования как агрегации, а не как ассоциации. Правда, если агрегация не представляет связь объектов достаточно хорошо, следует использовать ассоциацию. Борьба между аргументами в пользу общей ассоциации и специальной агрегации часто становится для проектировщика источником мучений.

Совместно используемую агрегацию можно реализовать на языке C++ подобно ассоциации, с указателями (или ссылками) на компонентные объекты. Композицию можно реализовать, используя объекты-части как элементы данных агрегированных объектов.

Обобщение представляет собой связь более общего и более конкретного класса. Более конкретный класс содержит те же атрибуты и операции, что и более общий класс, и может включать дополнительную информацию: атрибуты или операции. Обобщение реализуется в объектно-ориентированных языках программирования как наследование. Обобщение — это связь "является" ("is a") между классами. Обратите внимание, что это связь классов, а не экземпляров объектов. Класс может быть наследником другого класса в отличие от экземпляра объекта.

Конкретный класс (подкласс) связи обобщения наследует все атрибуты, операции и ассоциации от общего класса (суперкласса) связи. Нотация UML для этой связи использует сплошную линию для указания связи между классами на диаграмме классов. Отличить в связи подкласс от суперкласса на диаграмме можно по небольшому незакрашенному треугольнику на конце линии связи, обозначающей суперкласс.

На рис. 14.5(c) показаны два класса `Circle` и `Cylinder`, связанные обобщением. В этом случае `Circle` интерпретируется как обобщение (суперкласс), а `Cylinder` — как специализация (подкласс).

Если класс используется как суперкласс для нескольких специализаций, то каждый класс представляется на диаграмме классов отдельно, а каждый класс специализации связывается с суперклассом отдельной связью с отдельным треугольником, обозначающим суперкласс. Обычно применяется только один треугольник, указывающий на суперкласс, и с ним связывается каждый подкласс. На рис. 14.5(d) представлен класс `Account`, который используется как обобщение, и два других класса, `SavingsAccount` и `CheckingAccount`, которые представляют различные специализации класса `Account`.

Если подкласс используется как обобщение для другого класса, этот другой класс становится его специализацией. Класс может быть наследником одного класса и применяться как базовый класс для другого класса. На диаграммах классов UML появляется древовидная иерархия наследования.

Основы UML: нотация для множественности

Большинство связей являются бинарными, т. е. два класса связываются между собой. В действительности это не так. Вспомним предшествующее обсуждение классов `Person`, `Car` и `Registration`. Это тернарная связь: она включает объекты трех классов. Трудности, с которыми вы сталкивались во время обсуждения этой связи, возникали из-за того, что тернарную связь пытались представить как набор бинарных связей.

UML допускает нотацию для тернарных связей, однако он не поддерживает нотацию для связей между объектами более чем трех классов. Даже если бы он и поддерживал ее, то при реализации связей вы столкнулись бы с тем, что в языке C++ поддерживаются только бинарные связи. Связь между двумя объектами устанавливается с помощью физического или концептуального указателя. Итак, связи, моделируемые на диаграмме классов UML, являются бинарными, они связывают объекты двух классов.

Иногда связь соединяет объекты, являющиеся экземплярами одного класса. Например, объект класса `Person`, который играет роль руководителя, может быть ассоциирован с объектом класса `Person`, выполняющим роль участника группы. В данном случае оба объекта принадлежат одному классу. Во многих книгах по UML содержатся примеры рефлексивной (или рекурсивной) связи. На практике более удобно описать руководителя с помощью класса `Supervisor` и члена группы в классе `TeamMember`. Полезно иметь два различных класса в модели, поскольку они выполняют различные обязанности. И если у них много общих свойств, всегда можно ввести класс `Person` как их общий базовый класс.

Поэтому большинство связей на диаграммах классов UML являются бинарными. Каждая связь соединяет два объекта двух различных классов — один объект на одном конце связи и другой объект на другом конце связи.

Иногда объект одного класса может быть связан более чем с одним объектом другого класса. Например, объект класса `Supervisor` может быть ассоциирован с несколькими объектами класса `TeamMember`. На диаграмме класса UML все еще будет одна связь между классами `Supervisor` и `TeamMember`, но потребуется использовать дополнительную нотацию UML для указания множественности.

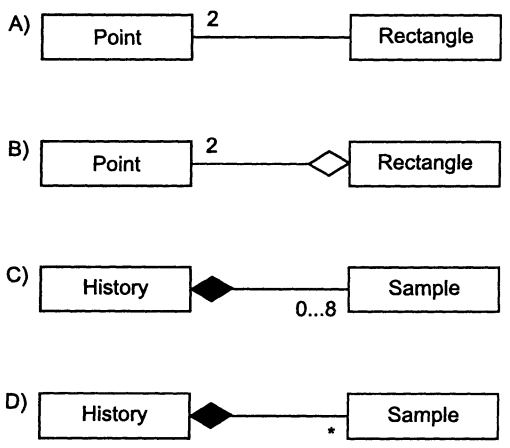


Рис. 14.6.
Примеры UML, показывающие множественность связей

На рис. 14.6 представлен пример обозначения множественности на диаграммах классов. На рис. 14.6(a) показаны два класса (`Point` и `Rectangle`) в приложении, где каждый объект `Rectangle` ассоциируется только с двумя объектами `Point`. Нотация UML, которая применяется к ассоциациям, может использоваться и для агрегаций. На рис. 14.6(b) показана связь между двумя классами, `Point` и `Rectangle`, которая интерпретируется скорее как агрегация, чем как общая ассоциация.

Обратите внимание, на рис. 14.3(c) в классе `Rectangle` у класса `Point` имеются два атрибута, `pt1` и `pt2`. Это означает, что любой объект класса `Rectangle` ассоциируется точно с двумя объектами класса `Point`. Следовательно, связь между классами на рис. 14.6(a) или 14.6(b) отображает ту же информацию анализа и проектирования, что и диаграмма классов. Некоторых специалистов огорчает эта избыточность, и они рекомендуют использовать только один способ для представления данной информации.

Советуем вам указать ассоциации и опустить атрибуты в классе. Логическое обоснование такого подхода заключается в том, что ассоциации представляют точку зрения анализа и проектирования, а атрибуты — точку зрения реализации. Поэтому на этапе анализа обозначаются связи, а при реализации они выражаются соответствующими указателями, элементами данных и т. д. Возможно, с практической точки зрения этот подход не столь важен. Поскольку не стоит беспокоиться о компиляторе UML, следуйте собственной интуиции.

Если в конец ассоциации или агрегации ничего не добавлено, значит, для функционирования связи требуется ровно один объект этого класса. На рис. 14.6(a) и 14.6(b) показано, что присутствие одного объекта класса `Rectangle` является обязательным.

Иногда связь между объектами не фиксируется, а ее множественность изменяется во время выполнения. Например, класс `History` в главе 12 ассоциирован с классом `Sample`. Фактически связь является композицией: объект класса `History` содержит массив объектов класса `Sample`. Как можно видеть из листинга 14.2, в массиве отсутствуют действительные объекты `Sample`. Во время выполнения программы появляются выборки измерений и информация сохраняется в массиве до тех пор, пока завершится программа либо количество объектов `Sample` будет равно восьми.

UML позволяет представить этот вид переменной множественности, указывая диапазон ассоциированных значений. На рис. 14.6(c) приведен пример, в котором количество ассоциированных объектов может изменяться от нуля до восьми. Если количество объектов не может быть меньше одного, то диапазон начинается с 1, а не с 0 (например, 1...8).

Часто диапазоны объектов в связи являются искусственными. Почему в `Sample` не может быть больше восьми объектов? Потому что язык C++ не допускает определение объекта без указания его длины. Число 8 подходит, как и любое другое число. В предметной области нет никаких указаний на то, что 8 лучше, чем 10, 20, 100 или любое иное число.

С концептуальной точки зрения количество выборок в истории не должно ограничиваться. По этой же причине реализация не должна заставлять проектировщика фиксировать конкретное число. В листинге 12.7 показан класс-контейнер с динамически выделяемой памятью, который реализует эту концептуальную модель. На рис. 14.6(d) представлено обозначение неограниченной множественности.

Учебный пример: магазин проката

Рассмотрим приложение, в котором ведется учет взятия напрокат и возврата фильмов клиентами в магазине проката видеофильмов.

Для простоты (предполагается, что магазин проката небольшой, а память компьютера велика) будем считать, что для выполнения приложения сначала в память компьютера загружается база данных клиентов и продукции, выдаваемой напрокат. Данные о сдаваемых напрокат фильмах включают наименование фильма, число имеющихся в магазине фильмов и идентификаторы фильмов. Данные о клиенте включают имя и номер телефона клиента (номер телефона используется в качестве идентификатора клиента), количество фильмов, взятых клиентом напрокат, и их идентификаторы.

Данный пример иллюстрирует основные вопросы проектирования классов, определения их связей и оптимизации проекта с точки зрения сведения до минимума зависимостей между классами.

Чтобы сделать пример более интересным с точки зрения использования наследования, добавлена следующая деталь: данные о фильмах сохраняются в файле с добавлением буквы, указывающей категорию фильма ("f" — для художественных фильмов, "c" — для комедий, "h" — для фильмов ужасов). Когда данные

считываются в память, информация о категории сохраняется в цифровой форме (1 — художественный фильм, 2 — комедия, 3 — фильм ужасов). При повторном сохранении данных в файле категория фильма снова сохраняется как буква.

Когда клиент приносит в магазин фильм для регистрации, продавец вводит номер телефона клиента для поиска в базе данных. Если клиента не находят, отображается сообщение. Если клиент есть в базе данных, отображаются имя и номер телефона клиента, а также данные о фильмах, которые он уже взял напрокат. После проверки имени клиента продавец вводит идентификатор фильма, количество имеющихся в наличии картин уменьшается на единицу, а идентификатор фильма добавляется в список идентификаторов фильмов, взятых этим клиентом напрокат.

Если для ввода идентификатора фильма используется считыватель штрих-кода, то фильм обязательно будет найден в базе данных. При вводе идентификатора фильма вручную, если фильм не найден, отображается сообщение об ошибке.

Когда клиент возвращает фильм, продавец снова вводит номер телефона клиента. При отображении учетной записи клиента продавец задает идентификатор фильма. Если вы находите идентификатор в списке фильмов, взятых клиентом напрокат, он удаляется из списка, а количество имеющихся в наличии фильмов увеличивается на единицу. При вводе неверного идентификатора фильма отображается сообщение об ошибке.

В этом примере не рассматривается финансовый аспект программы (взимание платы за прокат и оплата штрафа за несвоевременный возврат фильмов), опущена часть программы, оценивающая эффективность (вычисление индикаторов спроса для каждого фильма), и часть управления программой (добавление, удаление и редактирование данных о клиентах и о фильмах).

Классы и их ассоциации

Список классов приложения часто составляется при анализе функциональных спецификаций или другого документа, который описывает интерфейс и поведение пользователя системы.

Некоторые специалисты рекомендуют составлять список всех существительных из описания системы в качестве хорошей отправной точки. Другие специалисты высмеивают этот подход, поскольку большая часть существительных описывает сущности, которые не дотягивают даже до уровня класса (номер телефона, наличное количество и т. д.), а позже используются скорее как атрибуты (элементы данных), а не как классы.

Одно из предостережений в отношении использования описания системы для построения модели заключается в том, что описание построено на сущностях, взаимодействующих с системой (например, клиент, продавец магазина, база данных). Цель моделирования состоит в создании реализации системы, которая включает классы, содержащие данные, и оперирует этими данными (например, `Customer`, `StoreClerk` и `Database`). Объекты из описания системы и классы из реализации системы могут иметь одинаковые имена, но они не идентичны.

Предположим, что модель классов должна включать следующие классы: `Item` (информация о фильме), `Customer` (информация о клиенте), `Inventory` (управление набором фильмов и совокупностью клиентов), `File` (управление базой данных фильмов и клиентов) и `Store` (управление интерфейсом пользователя и запрос сервисов других классов). На рис. 14.7 показаны эти классы с указанием их атрибутов и операций.

Какие классы здесь связаны друг с другом? Следует признать, что изложенная система описания не очень помогает разгадать это. С другой стороны, не стоит торопиться признавать свою ошибку. Описание обычно создается, чтобы помочь в формировании и тестировании программы, а не для того, чтобы облегчить рисование моделей UML для системы.

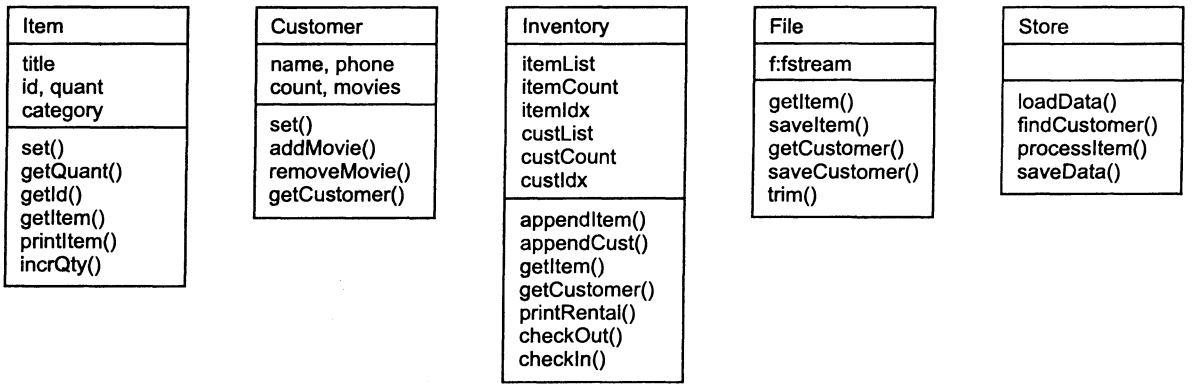


Рис. 14.7. *Нотация UML для классов с атрибутами и операциями*

Предполагается, что лучший способ обучения процессу создания моделей классов — это попытка задать несколько альтернативных вариантов для простого приложения, реализация каждого альтернативного варианта и оценка каждой реализации с точки зрения его сложности. Хорошим средством для такого типа обучения является упрощение приложения.

Однако в большинстве книг по объектно-ориентированному анализу и проектированию приводятся примеры диаграмм классов, используется описание системы как исходной точки без последующей реализации и, что более важно, без оценки того, как модель влияет на решения. Между тем решения о том, как распределить атрибуты и операции между классами и как связать классы, влияют на сложность программ.

Классы `Item` и `Customer` выполняют роль серверов для других классов. Они представляют такие услуги, как сохранение и извлечение значений элементов данных.

В проектах с несколькими файлами спецификация каждого класса помещается в отдельный заголовочный файл. Заголовочные файлы включаются в исходные файлы, которые реализуют клиентов класса, т. е. в исходные файлы, использующие имя класса для определения его переменных или параметров. В листинге 14.6 представлен заголовочный файл для класса `Item`, с его элементами данных и функциями-членами. Класс обеспечивает элементы данных для заголовка фильма, идентификатора, количества, имеющегося в наличии, и категории. Методы позволяют клиентской программе установить значения элементов данных объекта `Item` и извлечь идентификатор, количество объектов и все четыре элемента данных. Они также разрешают клиентской программе напечатать данные в требуемом формате (без имеющегося в наличии количества) и увеличить (или уменьшить) число фильмов, имеющееся в наличии.

Листинг 14.6. Спецификация класса для класса `Item` (файл `item.h`).

```
// file item.h

#ifndef ITEM_H
#define ITEM_H

class Item
{  protected:
          char title[26];
          int id, quant, category;
   public:
          void set (const char *s, int num, int qty, int type);
          int getQuant() const;
          int getId() const;
```

```
        void getItem(char* name, int &num, int& qty, int &type) const;
        void printItem() const;
        void incrQty(int qty);
    } ;
#endif
```

Обратите внимание на использование директив условной компиляции. В соответствии с правилами, унаследованными из языка C, заголовочный файл можно включать в исходные файлы своей программы только один раз. Если заголовочный файл включается более чем в один исходный файл, определение типа класса будет компилироваться в каждом исходном классе. Поскольку каждый исходный файл может компилироваться отдельно и в отдельный объектный файл, программа обрабатывается несколько раз, а компоновщик отбрасывает дополнительные определения, имеющие одинаковую структуру. Именно поэтому *каждый* программист на C++ должен помещать эти директивы условной компиляции в *каждый* заголовочный файл. Как жаль!

В листинге 14.7 представлен заголовочный файл для класса `Customer`. Здесь показан тот же набор директив условной компиляции для препроцессора, что и в листинге 14.6. Класс `Customer` предоставляет элементы данных для хранения имени клиента, номера телефона, счетчика фильмов, выданных клиенту напрокат, и идентификатора каждого взятого фильма. Его функции-члены позволяют клиентской программе установить значения для имени клиента и номера телефона, добавить идентификатор фильма в список фильмов, удалить идентификатор фильма из списка фильмов и извлечь имя клиента, номер телефона и список фильмов, взятых клиентом напрокат.

Листинг 14.7. Спецификация класса для класса `Customer` (файл `customer.h`).

```
// file customer.h

# ifndef CUSTOMER_H
# define CUSTOMER_H

class Customer
{  char name[20], phone[15];
   int count;
   int movies[10];
public :
   Customer ();
   void set(const char *nm, const char *ph);
   void addMovie(int id);
   int removeMovie(int id);
   void getCustomer(char *nm, char *ph, int &cnt, int m[]) const;
 } ;

# endif
```

Подобно заголовочным файлам, исходная программа на C++ для каждого класса в многофайловом проекте реализуется в отдельном исходном файле. В листинге 14.8 представлена реализация класса для класса `Item`. Заголовочный файл `<item.h>` должен быть включен в этот файл, чтобы убедиться, что компилятору известно значение оператора области действия `Item::`.

Реализация указывает, что класс `Item` для поддержки своей программы не требует каких-либо других классов. Ему нужны библиотечные средства. Некоторые проектировщики в свои диаграммы UML включают библиотечные компоненты как серверы классов. Вас же интересуют связи между компонентами вашей программы, а не то, как программа использует библиотечные классы и функции.

Листинг 14.8. Реализация класса `Item` **(файл** `item.cpp`**)**

```
// file item.cpp

#include <iostream>
using namespace std;
#include "item.h"                              // это необходимость

void Item: : set (const char *s, int num, int qty, int type)
{ strcpy(title,s); id=num; quant=qty; category=type; }

int Item::getQuant() const                     // используется Inventory::checkOut()
{ return quant; }

int Item::getId() const
{ return id; }                                 // в printRental(), checkOut(), checkIn()

void Item::getItem(char* name, int &num, int& qty,
         int &type) const                      // используется File::saveItem()
{ strcpy(name,title); num = id;
  qty = quant; type = category; }

void Item: :printItem() const                  // используется printRental()
{  cout.setf(ios::left, ios::adjustfieid) ;
   cout.width(5) ; cout << id;                 // ему известны его форматы вывода на печать
   cout.width(27) ; cout << title;
      switch (category) {                      // другие подтипы компонента
          case 1: cout << " feature"; break;
          case 2: cout << " comedy"; break;
          case 3: cout << " horror"; break; }
      cout << endl; }

      void Item::incrQty(int qty)              // используется в checkOut(), checkIn()
        { quant += qty; }
```

Для облегчения трассировки связей между классами используются построчные комментарии для указания места, откуда вызывается каждый метод `Item`. В комментариях указывается, что класс `Item` является сервером классов `Inventory` и `File`.

Подобным образом в листинге 14.9 представлен файл реализации для класса `Customer`. Заголовочный файл "`customer.h`" включается в этот файл. Заголовок любого файла реализации должен включаться в дополнение к заголовочным файлам для всех серверных классов, которые использует этот класс.

Листинг 14.9. Реализация класса `Customer` **(файл** `customer.cpp`**)**

```
// file customer.cpp

#include <iostream>
using namespace std;
#include "customer.h"                          // это необходимость

Customer::Customer()
{ count = 0; }

void Customer::set(const char *nm, const char *ph)
{ strcpy(name,nm); strcpy(phone,ph); }         // в appendCust()

void Customer::addMovie(int id)
{ movies[count++] = id; }                      // в appendCust(), в checkOut()
```

```
int Customer::removeMovie(int id)                    // используется в checkIn()
{  int idx;
   for (idx=0; idx < count; idx++)                   // найти фильм
     if (movies[idx] == id) break;
   if (idx == count) return 0;                       // возвратится, если не найден
   while (idx < count - 1)
   {  movies[idx] = movies[idx+1];                   // сдвиг остатка влево
     idx++; }
   count-;                                           // уменьшить счетчик фильмов
   return 1; }                                       // сообщить об успешном выполнении

void Customer::getCustomer(char *nm, char *ph,       // saveData()
   int &cnt, int m[]) const                          // Inventory::getCustomer()
{  strcpy (nm, name); strcpy(ph,phone); cnt = count;
   for (int i=0; i < count; i ++)
     m[i] = movies [i]; }
```

Видно, что файл исходной программы "customer.cpp" не включает какие-либо заголовочные файлы. Это означает, что класс Customer не содержит серверные классы — он сам обслуживает другие классы. Строковые комментарии в каждой функции указывают, когда функция используется как сервер для обеспечения клиентской программы с доступом к клиентским данным и сервисам.

Конструктор Customer инициализирует счетчик взятых напрокат фильмов, устанавливая его в нуль. Метод set() назначает новые значения для имени клиента и номера телефона, а метод addMovie() добавляет новый идентификационный номер в конец списка фильмов, предоставляемых напрокат.

Метод removeMovie() проверяет, находится ли идентификатор фильма в списке фильмов клиента. Если идентификатор в списке отсутствует, функция возвращает нуль, чтобы сообщить о сбое. Если идентификатор в списке имеется, метод сдвигает оставшиеся идентификационные номера на одну позицию влево, уменьшает счетчик допустимых идентификаторов фильмов и возвращает 1, чтобы подтвердить успешное выполнение.

Обратите внимание на то, что уменьшается счетчик идентификаторов, а не количество значений в массиве. Именно поэтому говорится о "счетчике действительных идентификаторов фильмов", а не о "счетчике идентификаторов фильмов".

Алгоритмы сдвига часто содержат ошибки, которые трудно обнаружить. Этот алгоритм легче понять, если уменьшить счетчик идентификаторов фильмов до выполнения сдвига влево, а не после него.

```
int Customer::removeMovie (int id)          // используется в checkIn()
{ int idx;
  for (idx=0; idx < count; idx++)           // найти фильм
    if (movies [idx] == id) break;
  if (idx == count) return 0;
  count-;                                   // уменьшить счетчик фильмов
  while (idx < count)                       // обычный вид
  {  movies[idx] = movies[idx+1];           // сдвиг остатка влево
    idx++; }
  return 1 ; }
```

Многие программисты пишут цикл сдвига в более краткой форме, используя оператор инкремента в операторе сдвига, а не помещая его в отдельной строке.

```
while (idx < count)
  movies[idx] = movies[idx++];              // кратко, но опасно
```

Вспомните, что присваивание в языке C++ является выражением, а в выражениях C++ гарантирует порядок оценки операций. Это правильно: порядок оценки компонентов в выражении не определен жестко (не гарантирован). Если выражение оценивается слева направо, то приведенный выше цикл работает прекрасно. Если выражение оценивается справа налево, то в цикле присутствует ошибка. Рекомендуем создавать более подробный код, который легче понять, чем краткий код, который запутывает программиста, осуществляющего сопровождение.

Как и в файле "item.cpp" в листинге 14.7, использованы комментарии к строкам, чтобы указать клиентов метода Customer. Эти комментарии показывают, что класс Inventory использует класс Customer в качестве сервера.

В листинге 14.10 представлен заголовочный файл для класса Inventory. Его элементы данных включают список фильмов и одного из клиентов, счетчики действительных компонентов в каждом списке и индексы для организации доступа к компонентам в каждом списке. Его функции-члены позволяют клиентской программе добавить фильм в список компонентов и клиента в список клиентов, извлечь текущий компонент из списка (указанный индексом itemIdx), извлечь текущий клиент из списка (указанного индексом custIdx), вывести на печать информацию, описывающую фильмы, взятые напрокат клиентом, отметить выдачу фильмов и зарегистрировать возврат фильма.

Листинг 14.10. Спецификации класса для класса Inventory **(файл** inventory.h**)**

```
// file inventory.h

# ifndef INVENTORY_H
#define INVENTORY_H
#include "item.h"
#include "customer.h"

class Inventory {
protected:
          enum { MAXM = 5, MAXC = 4 } ;                    // только для прототипа
          Item itemList[MAXM];
          Customer custList[MAXC];
          int itemCount, custCount;
          int itemIdx, custIdx;
public:
     Inventory ();
     void appendItem (const char* ttl, int id, int qty, int cat);
     void appendCust (const char* nm, const char* ph,
                      int cnt, const int *m);
     int getItem(Item& item);
     int getCustomer(char* nm, char* ph, int &cnt, int *m);
     void printRental(int id);
     int checkOut(int id);
     void checkIn(int id);
 } ;

#endif
```

Поскольку класс Inventory является клиентом классов Item и Customer, заголовочный файл Inventory должен включать заголовочные классы Item и Customer.

Некоторые программисты чувствуют себя незащищенными, Поэтому "на всякий случай" включают все заголовочные файлы проекта в каждый файл реализации. Как они говорят, лучше быть в безопасности, чем потом сожалеть.

Это неправильно. Вредно включать больше, чем нужно. Если что-то не включено, то компилятор пометит строки, использующие неопределенные имена, как содержащие ошибки. Когда включено больше, чем необходимо, исключается риск появления сообщения об ошибке. Однако для программиста клиентской части восприятие программы затрудняется.

Компилятор игнорирует избыточные определения в type. Читатели также не обратят на них внимание, но только после проверки программы класса и при обнаружении, что эти имена в type не используются в классе.

В данном случае можно поддержать такие рискованные действия (особенно потому, что при исключении ненужных заголовочных файлов риск отсутствует). Включение дополнительных заголовочных файлов "на всякий случай" не очень хорошая практика. Вместо того чтобы избежать синтаксических ошибок, можно еще больше запутать читателей.

Реализация класса Inventory представлена в листинге 14.11. Можно видеть, что включен только заголовочный файл "inventory.h". В этом файле используются объекты типа Item и Customer, но компилятор не выдаст сообщения о том, что данный тип имен неизвестен. В силу того факта, что заголовочные файлы Item и Customer включены в файл "inventory.h", они также входят и в реализацию класса Inventory. Подобно листингам 14.6 и 14.8, в него включены комментарии к строкам, которые указывают, из какой части серверной программы вызывается каждый метод. В отличие от классов Item и Customer класс Inventory содержит только один клиентский класс — класс Store.

Листинг 14.11. Реализация класса Inventory **(файл** inventory.cpp**)**

```
// file inventory.cpp

#include <iostream>
using namespace std;
#include "inventory.h"                              // это необходимость

Inventory::Inventory()
   { itemCount = itemIdx = 0; custCount = custIdx =0; }

void Inventory::appendItem (const char* ttl, int id,
                            int qty, int cat)
{ if (itemCount == MAXM)                            // используется в loadData()
    { cout << "\nNo space to insert item"; }
  else
    { itemList[itemCount++].set(ttl,id,qty,cat); } }

void Inventory::appendCust (const char* nm, const char* ph,
                            int cnt, const int *movie)
{ if (custCount == MAXC)                            // используется в loadData()
    { cout << "\nNo space to insert customer"; return; }
  custList[custCount++].set(nm,ph);
  for (int j=0; j < cnt; j++)
    custList[custCount-1].addMovie(movie[j]); }

int Inventory: :getItem(Item &item)                 // используется в saveData()
{ if (itemIdx == itemCount)
    { itemIdx = 0; return 0; }
  item = itemList[itemIdx++];
  return 1; }

int Inventory::getCustomer(char* nm, char* ph, int &cnt, int *m)
{ if (custIdx == custCount)                         // в findCustomer(), saveData()
    { custIdx = 0; return 0; }
  custList[custIdx++].getCustomer(nm,ph,cnt,m);
  return 1; }
```

```
void Inventory::printRental(int id)                    // используется в findCustomer()
{ for (itemIdx = 0; itemIdx < itemCount; itemIdx++)
      { if (itemList[itemIdx].getId() == id)
          { itemList[itemIdx].printItem(); break; } }
  itemIdx = 0;}

int Inventory::checkOut(int id)                        // используется в processItem()
{ for (itemIdx = 0; itemIdx < itemCount; itemIdx++
    if (itemList[itemIdx].getId() == id) break;
  if (itemIdx == itemCount)
  { itemIdx = custIdx = 0; return 0; }
  if (itemList[itemIdx].getQuant() == 0)
  { itemIdx = custIdx = 0; return 1; }
  itemList[itemIdx].incrQty(-1);
  custList [custIdx - 1 ].addMovie (id);
  itemIdx = custIdx = 0;
  return 2; }

void Inventory::checkIn(int id)                        // используется в processItem ()
{ if (custList[custIdx - 1].removeMovie(id) == 0)
    { cout << " Movie is not found\n";
      itemIdx = custIdx = 0; return; }
 for (itemIdx = 0; itemIdx < itemCount; itemIdx++)
    { if (itemList[itemIdx].getId() == id)
        { itemList[itemIdx].incrQty(1); break; } }
 itemIdx = custIdx = 0;
 cout << " Movie is returned\n"; }
```

Конструктор Inventory инициализирует индексы и счетчики компонентов (фильмов) и клиентов. Первоначально оба списка были пустыми. Методы appendItem() и appendCust() просты. Они тестируют имеющееся свободное пространство (этот тест соответствует прототипу, но избыточен при динамическом управлении памятью), добавляют компонент в конец массива и увеличивают счетчик действительных компонентов.

Методы getItem() и getCustomer() извлекают данные объекта из массива по указанному индексу (itemIdx для объекта Item, custIdx для объекта Customer). В одном случае извлекается весь объект, а в другом — значения элементов данных объекта. Поэтому в одном случае это клиентская программа, изменяющая значения элементов набора данных, а в другом случае представлен класс Inventory, который все выполняет по поручению клиентской программы.

Метод printRental() использует идентификатор фильма для поиска в массиве itemList[]. Если фильм найден, объекту посылается сообщение printItem().

Метод checkOut() с идентификатором фильма в качестве параметра осуществляет поиск компонента в массиве itemList[]. Если компонент не обнаружен, работа завершается и возвращается 0. Если компонент найден в данный момент, работа завершается возвращается 1. Если компонент доступен, количество имеющихся в наличии компонентов уменьшается на единицу, в список фильмов, взятых напрокат клиентами, добавляется идентификатор фильма и возвращается 2.

В методе checkIn() идентификатор фильма также используется как параметр. Поиск компонента в списке фильмов, взятых напрокат клиентами, осуществляется с помощью вызова метода removeMovie(). Если фильм не найден, checkIn() выводит на печать сообщение и завершает работу. Если фильм находится в списке клиента (а затем удаляется из списка), checkIn() ищет компонент в массиве компонентов itemList[], увеличивает значение имеющихся в наличии фильмов и выводит на печать подтверждение.

Интерфейсы методов checkIn() и checkOut() несовместимы. Метод checkOut() не вовлечен в диалог интерфейса пользователя. Вместо этого он возвращает значение, которое должен проанализировать клиент, и выводит на печать соответствующее сообщение. Работа передается клиентской части. Метод checkIn() отвечает за анализ состояний с ошибкой и соответствующий пользовательский интерфейс. Он скрывает состояния ошибки от клиентской части и возвращает допустимое значение.

```
Splash                  101  11  c
Birds                   102  22  h
Gone with the wind      103  10  f
```

Рис. 14.8.
Пример входного файла с данными о фильмах

Класс File разработан для осуществления доступа к физическим файлам, которые содержат данные о фильмах и клиентах до и после выполнения программы. На рис. 14.8 представлен пример файла ввода с данными о фильмах. Каждая строка в нем соответствует одному компоненту и включает заголовок фильма (с выравниванием по левому краю), идентификационный номер, число имеющихся в наличии фильмов и категорию (буква).

На рис. 14.9 можно видеть пример входного файла с данными клиента. Для каждого клиента выделяются две строки. Первая строка содержит имя и номер телефона клиента. Во второй строке хранится количество фильмов, взятых напрокат данным клиентом, и список номеров доступа к фильмам.

```
Shtern                  353-2566
2 101 102
Shtern                  358-0008
0
Simons                  277-7506
3 102 101 103
```

Рис. 14.9.
Пример входного файла с данными о клиентах

Входной и выходной файлы имеют одинаковый формат. Информационное содержание файла вывода для компонентов, полученное в результате выполнения программы показано на рис. 14.10. Вы видите, что была возвращена одна копия "Splash" и выдана одна копия "Gone with the Wind".

```
Splash                  101   12   c
Birds                   102   22   h
Gone with the wind      103    9   f
```

Рис. 14.10.
Пример выходного файла с данными о фильмах

На рис. 14.11 представлено содержание файла с данными о клиентах после выполнения программы. В нем указывается, что клиент Штерн возвратил фильм с идентификационным номером 101 и взял напрокат фильм с идентификационным номером 103.

В листинге 14.12 приведены спецификации класса для класса File. Этот класс инкапсулирует файловый объект fstream, способный выполнять считывание и запись данных. Класс реализует общедоступные методы getItem() и saveItem(), выполняющие операции ввода/вывода для данных Item. Он также реализует общедоступные методы getCustomer() и saveCustomer(), которые выполняют операции ввода/вывода для данных Customer.

```
Shtern                  353-2566
2   101   102
Shtern                  358-0008
0
Simons                  277-7506
3   102   101   103
```

Рис. 14.11.
Пример выходного файла с данными о клиентах

Листинг 14.12. Спецификации класса для класса File (файл file.h)

```
// file file.h

#ifndef FILE_H
#define FILE_H
#include "item.h"
#include <fstream>

class File
{  fstream f;
   static void trim(char buffer []);
   enum { TWIDTH = 27, IWIDTH = 5, QWIDTH = 6,
          NWIDTH = 18, PWIDTH = 16 };
public:
   File(const char name[], int mode);
   int getItem(char *ttl, int &id, in &qty, char &type);
   void saveItem(const Item &item);
```

```
  int getCustomer(char *name, char *phone, int &count, int *m);
  void saveCustomer(const char *nm, const char *ph,
    int cnt, int *m) ;
} ;

#endif
```

В листинге 14.13 представлена реализация класса File. Его конструктор открывает физический файл либо для чтения, либо для записи и тестирует проведение операции с помощью функции Fail(). Другой способ протестировать проведение операции — вызвать функцию is_open(), которая возвращает true, если файл успешно открылся.

Листинг 14.13. Реализация класса File (файл file.cpp)

```
// file file.cpp

# include <iostream>
using namespace std;
# include "file.h"                                  // это необходимость

File::File(const char name[], int mode)
{  f.open (name,mode);                              // используется в loadData(), saveData()
   if (f.fail())                                    // также, если (f.is_open()) успешно
     { cout << " File is not open\n" ; exit(1); } }

int File:: getItem(char *ttl, int &id, int &qty, char &type)
{  char buffer [200];                               // в loadData()
   f.get(buffer,TWIDTH);
   trim(buffer) ;
   strcpy(ttl,buffer);                              // знает структуру файла
   f >> id; f >> qty; f >> type ; f.getline(buffer,4);
   if (!f) return 0;
   return 1; }

void File::saveItem(const Item &item)               // в saveData()
{  char tt[27]; int id, qty, type;
   item.getItem(tt,id,qty,type) ;
   f.setf(ios::left, ios::adjustfield);
   f.width (TWIDTH) ; f << tt;                      // знает формат файла
   f.setf (ios: : right, ios: : adjustfield);
   f.width(IWIDTH); f << id ;
   f.width (QWIDTH) ; f << qty;
   switch (type) {                                  // отличается от других подтипов
     case 1: f << " f\n"; break;
     case 2: f << " c\n"; break;
     case 3: f << " h\n"; break; } }

int File::getCustomer(char *name, char *phone, int &count, int *m)
{  char buffer[200] ;                               // в loadData()
   f.get(buffer, NWIDTH);
   trim (buffer);
   strcpy(name,buffer);
   f >> buffer; f >> count;                         // знает структуру файла
   strcpy(phone,buffer);
   for (int i=0; i < count; i++)
     f >> m [i];
   f.getline(buffer,2);
```

```
   if (!f) return 0;
   return 1; }

void File::saveCustomer(const char *nm, const char *ph,
                        int cnt, int *m)        // в saveData()
{ f.setf(ios::left,ios::adjustfield); f.width(NWIDTH);
   f << nm;
   f.setf(ios::right,ios::adjustfield); f.width(PWIDTH);
   f << ph << endl << cnt;                      // знает структуру файла
   for (int i=0; i < cnt; i++)
   { f .width(6); f << m [i]; }
   f << endl; }

void File::trim(char buffer[])                  // в getItem(), getCustomer()
{ for (int j = strlen(buffer)-1; j>0; j-)
      if (buffer[j]==' '||buffer[j]=='\n')
         buffer[j] = '\0';
      else
         break; }
```

Метод `getItem()` считывает одну строку данных из входного файла в локальный массив `buffer[]`, отбрасывает конечные пробелы и копирует данные в выходной массив `ttl[]`. Затем он считывает данные из файла в другие компоненты данных элемента (идентификационный номер, имеющееся в наличии количество, категория). Окончательный вызов `getline()` порождает выдачу условия конца файла (end of file), если только что считанная строка является последней строкой физического файла. В этом случае файловый объект становится нулевым, а `getItem()` возвращает нуль, чтобы указать конец входных данных для вызывающей программы (класс `Store`). Иначе возвращается единица, показывающая, что еще имеются данные для чтения.

Метод `saveItem()` сохраняет данные элемента в физическом файле. Убедиться, что категория целого типа правильно преобразуется в соответствующий символьный тип, можно с помощью оператора `switch`.

Метод `getCustomer()` считывает имя клиента, отбрасывает конечные пробелы, считывает номер телефона клиента и количество фильмов, взятых напрокат, а затем идентификаторы взятых напрокат фильмов.

Метод `saveCustomer()` записывает в физический файл имя клиента, номер телефона, счетчик фильмов и идентификаторы фильмов.

Метод `trim()` удаляет конечные пробелы из имени, потому что `getline()` не останавливается, обнаружив конец слова во входном файле. Иногда бывает нужно указать количество считываемых символов либо признак конца (возврат каретки). Строка, в которой удаляются конечные пробелы, передается как параметр. Метод `trim()` не затрагивает другие элементы данных класса. Следовательно, функция `trim()` должна быть объявлена статической. Метод `trim()`, выполняющий отбрасывание конечных пробелов, вызывается только из методов `File`: `getItem()` и `getCustomer()`. Функция `trim()` должна объявляться закрытой.

В этом проекте классом верхнего уровня является класс `Store`. В листинге 14.14 представлены его спецификации. Класс `Store` — сервер только одного программного компонента, глобальной функции `main()`, однако все равно рассмотрим условную компиляцию.

Этот файл не будет компилироваться без включения заголовочного файла "`inventory.h`", поскольку компилятор не будет знать, что означает имя `Inventory`. Однако можно скомпилировать его без заголовочного файла "`file.h`", поскольку имя класса `File` упоминается только в реализации функций-членов класса `Store` (см. листинг 14.15).

Листинг 14.14. Спецификации класса для класса `Store` **(файл** `store.h`**)**

```
// file store.h

#ifndef STORE_H
#define STORE_H
#include "inventory.h"
#include "file.h"

class Store {
public:
  void loadData(Inventory &inv);
  int findCustomer(Inventory& inv);
  void processItem(Inventory& inv);
  void saveData(Inventory &inv);
} ;

#endif
```

Следовательно, вы можете включить заголовочный файл "`file.h`" в файл реализации, а не в заголовочный файл для класса `Store`. Компилятор не столкнется с трудностями при вычислении. Возможно, для пользователя это не очень хорошая идея. Лучше все серверные заголовочные файлы хранить в одном месте, в заголовочном файле класса. Тогда программист, осуществляющий сопровождение, сможет сразу увидеть все серверные классы, используемые данным классом.

Некоторые проектировщики включают заголовочные файлы для серверов в серверы, например "`item.h`" и "`customer.h`". Правда, из-за этого создается неразбериха в клиентских заголовочных классах.

Как показано в листинге 14.14, класс `Store` не содержит элементов данных. Это могло бы вызвать тревогу для класса в середине иерархии классов, но нормально для клиентского класса верхнего уровня. Методы класса `Store` отвечают за операции верхнего уровня, которые описывают внешние интерфейсы системы: за загрузку базы данных в начале работы системы, поиск клиента в базе данных, обработку запросов напрокат фильмов клиентами и за сохранение базы данных после завершения программы.

В листинге 14.15 приведена реализация класса Store. Метод `loadData()` создает локальный объект класса `File` и отправляет ему сообщения `getItem()` для считывания данных с внешнего файла. Каждый набор данных `Item` используется как аргумент в вызове `appendItem()`. Это сообщение отправляется объекту `Inventory`, а `loadData()` получает его как параметр. Затем `loadData()` создает другой локальный объект класса `File`, считывает клиентские данные из файла и сохраняет их в объекте `Inventory`. Локальный объект класса `File` исчезает, когда завершается `loadData()`. При этом разрывается связь между физическими файлами "`Item.dat`" и "`Cust.dat`" и объектами `File`.

Листинг 14.15. Реализация класса `Store` **(файл** `store.cpp`**)**

```
// file store.cpp

#include <iostream>
using namespace std;
#include "store.h"                                  // это необходимость

void Store::loadData(Inventory &inv)
{ File itemsIn("Item.dat",ios::in);                 // компонент базы данных
  char ttl[27], category; int id, qty, type;        // компонент данных
  cout << "Loading database ... " << endl;
```

```
   while (itemsIn.getItem(ttl,id,qty,category)== 1)      // считывание
   { switch (category) {                                  // определение категории для подтипа
        case 'f': type = 1; break;
        case 'c': type = 2; break;
        case 'h': type = 3; break; }
     inv.appendItem(ttl,id,qty,type); }
   File custIn("Cust.dat",ios::in);                       // база данных клиента
   char name[25], phone[15]; int movies[10], count;
   while (custIn.getCustomer(name,phone,count,movies) == 1)
   { inv.appendCust(name,phone,count,movies); } }         // скачивание данных

int Store::findCustomer(Inventory& inv)
{  char buffer[200]; char name[25], phone[13];
   int count, movies[10];
   cout << "Enter customer phone (or press Return to quit) ";
   cin.getline(buffer,15) ;
   if (strcmp(buffer,"") == 0) return 0;                  // выход при отсутствии ввода данных
   bool found = false;
   while (inv.getCustomer(name,phone,count,movies) != 0)
      { if (strcmp(buffer,phone) == 0)                    // поиск номера телефона
          { found = true; break; } }                      // останов, если телефон найден
   if (!found)
   { cout << "\nCustomer is not found" << endl;
     return 1; }                                          // продолжение, если не найден
   cout.setf(ios::left,ios::adjustfield);
   cout.width(22); cout << name << phone << endl;         // вывод на печать данных
   for (int j = 0; j < count; j++)
     { inv.printRental(movies[j]);}                       // печать идентификатора фильма
   cout << endl;
   return 2; }                                            // код успешного выполнения

void store: :processItem(Inventory& inv)
   { int cmd, result, id;
     cout << " Enter movie id: ";
     cin >> id;                                           // поиск атрибута
     cout << " Enter 1 to check out, 2 to check in: ";
     cin >> cmd;
     if (cmd == 1)
        {  result = inv.checkOut(id);                     // анализ возвращенного значения
           if (result == 0)                               // не найден
              cout << "Movie is not found " << endl;
           else if (result == 1)                          // нет в запасе
              cout << "Movie is out of stock" << endl
           else                                           // успешно
              cout << " Renting is confirmed\n"; }
     else if (cmd == 2)
       inv.checkIn(id);                                   // обратная связь в checkIn()
     cin.get(); }                                         // исключение CR из строки

void Store::saveData(Inventory &inv)
{  File itemsOut("Item.out",ios ::out); Item item;       // файл компонента
   while (inv.getItem(item))                              // отсутствует внутренняя структура
     itemsOut.saveItem(item);                             // сохранение каждого компонента
   File custOut ("Cust.out",ios::out) ;                   // выходной файл клиента
   char name[25], phone[13]; int count, movies[10];
   cout << "Saving database ... " << endl;
   while(inv.getCustomer(name,phone,count,movies))    // скачивание данных
     custOut.saveCustomer(name,phone,count,movies);}
```

Метод findCustomer() запрашивает номер телефона клиента и завершается (возвращая нуль), если оператор нажал клавишу Enter без ввода каких-либо данных. Если вводится номер телефона, то findCustomer() извлекает данные клиента, отправляет сообщение getCustomer() объекту Inventory, который пересылает его findCustomer() как аргумент. Если номер телефона не находится, выводится сообщение об ошибке и findCustomer() возвращает 1 для уведомления клиента. В противном случае выводятся имя клиента, номер телефона и данные о фильме, а метод возвращает 2.

В методе processItem() также имеется параметр типа Inventory. Метод запрашивает у оператора ввод идентификатора фильма и команду (для регистрации выдачи или возврата), а затем отправляет либо сообщение checkOut(), либо checkIn() его параметру. Когда возвращается checkOut(), processItem() анализирует возвращенное значение и выводит на печать следующее сообщение. Когда возвращается checkIn(), processItem() просто завершается, потому что checkIn() анализирует результаты операции и выводит на печать сообщения оператору.

Метод saveData() точно повторяет действия loadData(). Он создает локальные объекты класса File и отправляет сообщения saveItem() и saveCustomer() с информацией, которая была извлечена saveData() из параметра Inventory. Используются сообщения getItem() и getCustomer().

Листинг 14.16. Реализация функции main() Store (файл video.cpp)

```
// file video.cpp

#include <iostream>
using namespace std;
#include "store.h"                          // это необходимость

int main()
{  Inventory inv; Store store;              // определение объектов
   store.loadData(inv) ;                    // загрузка данных
   while(true)
      { int result = store.findCustomer(inv);   // проверка результатов
        if (result == 0) break;             // завершение программы
           if (result == 2)                 // 1, если не найден
              store.processItem(inv); }     // обработка кассеты
   store.saveData(inv);                     // сохранение базы данных
   return 0;
}
```

Последним компонентом программы является клиент Store функции main(), реализующей два объекта (см. листинг 14.16), один из класса Inventory, а другой из класса Store. Последний отправляет сообщения объекту Store и передает объект Inventory как аргумент.

На рис. 14.12 показан пример выполнения программы. Он соответствует входным файлам на рис. 14.8 и 14.9. Выходные файлы, сгенерированные при выполнении программы, представлены на рис. 14.10 и 14.11.

Предположительно список классов, реализуемых приложением, соответствует списку объектов реального мира, с которыми имеет дело система. Обычно распределение обязанностей в приложении между классами достаточно естественное. Закономерно, что класс Item сохраняет информацию о фильмах, а не имена клиентов или файлы дисков.

```
Loading database ...

Enter customer phone (or press Return to quit) 353-2566
Shtern                353-2566
101  Splash                          comedy
102  Birds                           horror

  Enter movie id: 101
  Enter 1 to check out, 2 to check in: 2
  Movie is returned
Enter customer phone (or press Return to quit) 353-2566
Shtern                353-2566
102  Birds                           horror

  Enter movie id: 103
  Enter 1 to check out, 2 to check in: 1
  Renting is confirmed
Enter customer phone (or press Return to quit)
Saving database ...
```

Рис. 14.12. *Примеры выполнения программы, приведенной в листингах 14.6–14.16*

Это понятно. Все становится менее определенными при переходе к клиентским классам на вершине иерархии классов. Класс Store не имеет каких-либо интуитивно понятных обязанностей. Разделение обязанностей между классом Store и main() совершенно произвольное. Некоторые проектировщики чувствуют, что main() должен создать экземпляр приложения для начального объекта. После вызова этого конструктора будут осуществляться остальные действия.

При таком подходе содержимое main() должно передаваться конструктору Store. Объект Store не нужен в конструкторе, поскольку функции-члены Store доступны в конструкторе немедленно, без целевого объекта. Поскольку функции-члены Store вызываются только из конструктора Store, они не должны быть общедоступными (public), они могут быть объявлены как закрытые.

```
Class Store {
private:
  void loadData(Inventory &inv);
  int findCustomer(Inventory& inv);
  void processItem(Inventory& inv);
  void saveData(Inventory &inv);
public:
  Store(void)
    {  Inventory inv;                          // определение объектов
       loadData(inv);                          // загрузка данных
       while (true)
         {  int result = findCustomer(inv) ;   // проверка результатов
            if (result == 0) break;            // завершение программы
            if (result == 2)                   // 1, если не найден
               processItem(inv); }             // обработка кассеты
       saveData(inv); }                        // сохранение базы данных
} ;
```

Функция main() становится совсем простой.

```
int main()
{ Store store;
  return 0; }
```

Как уже упоминалось ранее, разделение обязанностей между начальными классами в иерархии классов и функцией main() является произвольным и не может быть спроектировано из анализа функциональных возможностей системы.

Видимость класса и разделение обязанностей

Рассмотрим связи классов.

В первой части книги обсуждалась идея разделения обязанностей между функциями, чтобы избежать чрезмерного взаимодействия между ними (и усиленного сотрудничества разработчиков). Избыточное общение часто происходит в результате разделения на части того, что должно составлять одно целое.

Здесь мы поговорим о разделении обязанностей между классами, чтобы избежать избыточного обмена сообщениями между классами и чрезмерного общения разработчиков, отвечающих за различные классы.

Избыточный обмен сообщениями между классами часто происходит в результате разделения на части целого, например, при разделении обязанностей между различными функциями и различными классами, так что они должны осуществлять связь через параметры функций и элементы данных класса. Чем обширнее передача сообщений между классами, тем больше подробностей должны помнить проектировщики. Повышается вероятность возникновения ошибок.

Кроме того, работа с классами включает и передачу обязанностей от клиентских классов серверным классам. Если вы не можете это сделать, появляются простые серверные классы, но усложняются клиентские классы и затрудняется их понимание. В результате программисту клиентской части и лицу, осуществляющему сопровождение, трудно выполнять поставленную задачу.

Дополнительная концепция, которая относится только к проектированию с использованием классов, а не к проектированию с функциями, это концепция видимости класса. Чем больше серверных классов использует клиентский класс, тем внимательнее должен быть проектировщик клиентской части и программист, осуществляющий сопровождение. Они должны изучить интерфейсы серверных классов и понять ограничения на использование серверных классов. Уменьшение количества серверных классов, видимых клиентскому классу (о котором проектировщик клиентской части должен знать), облегчает понимание программы.

Наоборот, чем больше клиентских классов используют один серверный класс, тем чувствительней разработка программы к изменениям серверного класса. Уменьшение числа клиентских классов, для которых виден серверный класс, повышает значимость программы. Любую программу можно спроектировать с использованием только одного класса (или вообще не используя классы), и проблемы обмена сообщениями между классами, разделения обязанностей между ними и видимости классов между собой исчезнут. Вам требуется построить программу с совместно работающими классами, однако взаимодействие классов должно быть минимальным.

Использование диаграмм классов UML (см. рис. 14.4) — хороший метод анализа структуры программы. Связи класса на диаграмме показывают, каким клиентским классам известно о конкретном серверном классе. К сожалению, связи классов на диаграммах UML не отображают разделение обязанностей между классами, передачу обязанностей к серверам и разделение на части того, что должно составлять одно целое. Для этого необходимо проанализировать распределение элементов данных и функций-членов по классам. Диаграммы классов (см. рис. 14.3) более полезны.

Видимость класса и связи классов

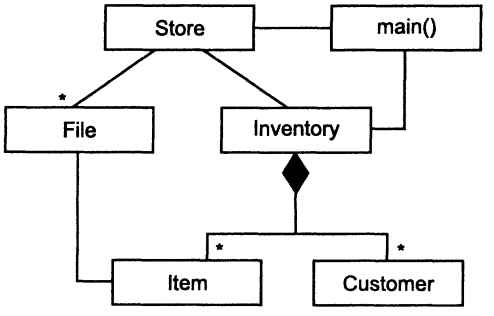


Рис. 14.13. *Диаграмма классов для программ, приведенных в листингах 14.6–14.16*

На рис. 14.13 показаны связи между классами, описанными в учебном примере в листингах 14.6–14.16.

Диаграмма классов UML позволяет понять, что класс `Inventory` "владеет" произвольным количеством объектов классов `Item` и `Customer`. В проекте класса `Inventory` были ограничены размеры массива, но они были произвольными и имели отношение к концептуальным связям между классом `Inventory` и содержащимися в нем объектами. С концептуальной точки зрения класс `Inventory` может содержать неограниченное число объектов `Item` и `Customer` (см. диаграмму классов на рис. 14.13).

Остальная часть диаграммы классов показывает, что класс `Store` является клиентом классов `Inventory` и `File` и что `main()` представляет собой клиент класса `Store` и класса `Inventory`. Она также показывает, что класс `File` является клиентом класса `Item`, но не класса `Customer`.

Возникает противоречие, о котором говорилось при проектировании классов `Item` и `Customer`. Объекты класса `Item` знают, как вывести себя на печать. Объекты же класса `Customer` с этим не знакомы. Именно поэтому класс `Customer` предоставляет метод `getCustomer()`, который используется клиентской программой для извлечения элемента данных `Customer` для вывода на печать.

Далее противоречие поддерживается структурой класса `Inventory`. Его метод `getItem()` обеспечивает клиентскую программу объектом `Item`. Программа осуществляет доступ к компонентам объекта `Item`. Метод `getCustomer()` класса `Inventory`

обеспечивает клиентскую программу компонентами Customer, но не объектом Customer. Именно поэтому класс File видит класс Item, а не класс Customer.

Видимость одного класса в другом классе этой же программы является важной характеристикой, которую проектировщики могут использовать для уменьшения до минимума зависимостей классов и координации проектировщиков.

Когда объект в клиентском методе определяется как локальный объект, он виден. Размер координации минимальный. Примером является класс File, объекты которого определяются только в методах loadData() и saveData() класса Store и не видны в других классах или в других методах класса Store.

Когда объект определяется как элемент данных в клиентском классе, его видят все методы клиентского класса. Это более сильная степень зависимости — клиентские методы должны координировать использование серверных объектов. Примером является класс Item и класс Customer. Их объекты задаются как элементы данных класса Inventory и индексов custIdx и itemIdx, которые обозначают эти объекты. Все методы класса Inventory имеют доступ к этим двум массивам и к индексам.

Рассмотрим, например, листинг 14.11, где представлена реализация класса Inventory. Метод getCustomer(), который вызывается из метода findCustomer() класса Store, устанавливает индекс custIdx, обозначающий объект Customer. Он будет участвовать в операциях регистрации выдачи напрокат и возврата фильма. Методы checkOut() и checkIn() осуществляют доступ к одному и тому же объекту и используют ту же переменную индекса custIdx. Однако они должны вычитать 1 для получения правильного объекта. Это пример связи, создаваемой посредством доступа к одному и тому же вычислительному объекту из различных методов.

Когда клиентский объект определяется в методе собственного клиента, его сервер можно отправить его методам как параметр. Например, на рис. 14.13 показано, что класс Store является клиентом класса Inventory. Клиентский объект (Store) определяется как локальная переменная его клиента (функция main()), а объект-сервер (Inventory) посылается методам как параметр.

В листинге 14.16 представлена реализация этой связи. Функция main() является клиентом обоих классов Inventory и Store. Она определяет объекты Inventory и Store и посылает объект Inventory методам Store как аргумент. Проектировщики main() и Store должны знать о классе Inventory.

Это хорошо знакомый вопрос о намеренном сокрытии информации (о деталях реализации) от пользователя, который можно обсудить с позиции видимости объекта. Если объект Inventory определяется как элемент данных класса Store, а не как переменная в main(), то речь идет о методах класса Store, которые имеют доступ к этому объекту.

```
class Store {
  Inventory inv;
public:
  void loadData();
  int findCustomer();
  void processItem();
  void saveData();
} ;
```

Принудительная передача обязанностей серверным классам

Принудительная передача обязанностей серверным классам является хорошим способом рационализации программы в клиентских методах и исключения деталей обработки нижнего уровня, которые затрудняют чтение клиентской программы и не позволяют быстро понять смысл обработки.

Например, в листинге 14.6 класс Item предусматривал методы getId() и getQuant(). Это общие методы, предоставляющие действительный идентификатор элемента и количество элементов. Вследствие такой общности подобный проект отвечает любым требованиям, предусматривающим использование этих данных.

Это хорошо в библиотечном классе, который желательно продать как можно большему количеству возможных клиентов. Но хуже в той части программы, которую требуется спроектировать для удовлетворения конкретных запросов, полученных от клиентских классов, принадлежащих к этой же программе или к ее следующей версии. С общей структурой "библиотечного типа" клиентские классы должны быть гибкими, чтобы использовать сервисы, которые обеспечивают серверные классы. Обычно клиентские классы получают от серверов намного больше информации, чем необходимо в действительности. Она должна отвечать текущим потребностям клиента.

В листинге 14.11 клиентская функция printRental() просматривает каждый объект Item в классе Inventory и возвращает значение идентификатора объекта Item. Теперь функция printRental() может делать с этим значением все, что ей угодно, но ей требуется только сравнить его со значением параметра.

```
void Inventory::printRental(int id)          // используется в findCustomer()
{ for (itemIdx = 0; itemIdx < itemCount; itemIdx++)
  { if  (itemList[itemIdx].getId() == id)
        { itemList[itemIdx].printItem(); break; } }
  itemIdx = 0;}
```

Эта информация избыточна, поскольку клиентской программе требуется знать только, является ли идентификатор в следующем объекте Item тем же, что и значение параметра. Клиентская программа получает больше информации, чем ей необходимо (значение идентификатора), но она должна много работать с этой информацией. При разделении обязанностей клиентская программа должна получить значение параметра серверной функции, тогда серверная программа сможет выполнить работу от имени клиента (сравнить идентификаторы). Клиентская программа будет иметь следующий вид:

```
void Inventory::printRental(int id)          // используется в findCustomer()
{ for (itemIdx = 0; itemIdx < itemCount; itemIdx++)
  { if  (itemList[itemIdx].sameId(id))       // важное отличие
        { itemList[itemIdx].printItem(); break; } }
  itemIdx = 0;}
```

Клиентская функция checkOut() в листинге 14.10 вызывает серверную функцию getQuanto(), чтобы решить, можно ли выдать напрокат данный фильм. Теперь клиентская функция может делать с этим значением, что требуется, но она просто сравнивает его с нулем.

```
int Inventory::checkOut(int id)              // используется в processItem()
{ for   (itemIdx = 0; itemIdx < itemCount; itemIdx++)
        if (itemList[itemIdx].getId() == id) break;
  if (itemIdx == itemCount)
  { itemIdx = custIdx = 0; return 0; }
  if (itemList[itemIdx].getQuant()==0)       // какое значение?
  { itemIdx = custIdx = 0; return 1; }
  itemList[itemIdx].incrQty(-1);
  custList[custIdx - 1].addMovie(id);
  itemIdx = custIdx = 0;
  return 2; }
```

Снова эта информация является избыточной, поскольку клиентской программе нужно знать только, имеется ли нужный компонент. Клиентская программа получает больше информации, чем необходимо, но она должна много работать с ней. При разделении обязанностей серверную функцию надо сравнить с нулем, так что клиентская программа даже не будет знать об используемых правилах проверки доступности компонента. Чтобы избежать передачи информации для обработки от серверной к клиентской программе, сервер может предоставить функцию inStock(). Клиентская программа будет выглядеть так:

```
int Inventory::checkOut(int id)                    // используется в processItem()
{ for  (itemIdx = 0; itemIdx < itemCount; itemIdx++)
       if (itemList[itemIdx].sameId(id)) break;
  if (itemIdx == itemCount)
  { itemIdx = custIdx =0; return 0; }
  if (itemList[itemIdx].inStock))                  // значение очевидно
  { itemIdx = custIdx = 0; return 1; }
  itemList[itemIdx].incrQty(-1);                   // задание передается серверу
  custList[custIdx-1].addMovie(id);                // задание передается серверу
  itemIdx = custIdx = 0;
  return 2; }
```

Обратите внимание, что функция checkOut() может сохранить значение количества фильмов, имеющихся в наличии, проверить, больше ли оно нуля, уменьшить его на 1 и сохранить новое значение количества в объекте Item. Это другой пример передачи обязанностей клиентской программе. Вместо этого функция checkOut() говорит объекту компонента: "Мне неизвестно, сколько здесь компонентов и не стоит беспокоиться о точном числе, поскольку я знаю, что в наличии имеются фильмы для выдачи напрокат". Это хороший пример передачи обязанностей от клиентского класса серверному классу.

Использование наследования

На диаграмме UML (см. рис. 14.13) наследование не используется, поскольку это скорее реализация метода, чем модель связи между объектами реального мира.

Наследование применяется для упрощения проекта серверных классов, программы клиентских классов и для уменьшения количества общей информации для классов в приложении.

Например, учебный пример в листингах 14.6–14.16 реализует некоторый вид идиосинкратического поведения компонентов Inventory. Во входном файле вид фильма обозначается буквой, например "f". Это же происходит и в выходном файле. В отображенном компоненте вид фильма указывается словом, например "художественный" (feature). В памяти во время выполнения он обозначается целым числом, например 1.

Это обычные требования. Убедитесь в том, что клиенты серверного класса защищены от подобных действий. Проект в листингах 14.6–14.16 не очень хорошо отвечает этим требованиям. Класс Item знает об этом в своем методе printItem(), он решает, какое слово отобразить. Так же поступает клиент Item — File: в своем методе saveItem() принимает решение, какую букву записывать в выходной файл. Аналогично действует класс Store в своем методе loadData(): Store проверяет, какое целое значение нужно сохранить в памяти компонента для последующего использования. И только класс Inventory не затрагивается в данном вопросе, поскольку по ошибке забыли включить проверку того, что он делает.

Если проектировщик не пытается сохранить общую информацию о классах, она распределяется вокруг программы.

Наследование — это хороший механизм для сохранения информации в серверных классах. Если класс Item станет базовым классом для набора производных классов, например FeatureItem, ComedyItem и HorrorItem, вы можете сохранить информацию о поведении компонента в этих классах и предотвратить ее расползание по программе.

В данной главе это решение не реализовано, поскольку для него требуется использование полиморфизма.

Другой вопрос, связанный с использованием наследования в учебном примере в листингах 14.6–14.16, заключается в проектировании класса File. В данной программе объекты класса File используются для четырех целей: чтения данных компонента (фильма), чтения данных клиента, записи данных компонента и записи клиентских данных. Каждый объект подходит только для одной цели. Например, объект itemsOut класса File в методе saveData() в листинге 14.15 может использоваться только для записи данных компонента. Если программист клиентской части попытается отправить сообщение getCustomer() класса File этому объекту, компилятор примет вызов данной функции. Но во время выполнения программа будет прекращена, потому что физический файл открыт для записи.

Обратите внимание, что если программист клиентской части использует этот объект класса File для приема сообщения saveCustomer(), то компилятор не только примет эту программу, но выполняемая система не выдаст возражений. Неверные данные будут записаны в выходной файл.

Использование наследования допускает создание специализированных классов, которые могут выполнять только один вид работы. Например, класс FileOutItem может записывать данные в файл, содержащий данные компонента. Он не может считывать данные или записывать данные о клиенте.

```
class FileOutItem : public File
{ public:
    FileOutItem(const char name[]);
    void saveItem( const Item &item);
      };
```

При такой структуре попытка клиентской программы отправить объекту FileOutItem сообщение getItem() или saveCustomer() будет интерпретирована компилятором как синтаксическая ошибка. Это очень хорошо. Альтернатива в том, чтобы ликвидировать в программе большое количество маленьких классов, и это может серьезно усложнить сопровождение.

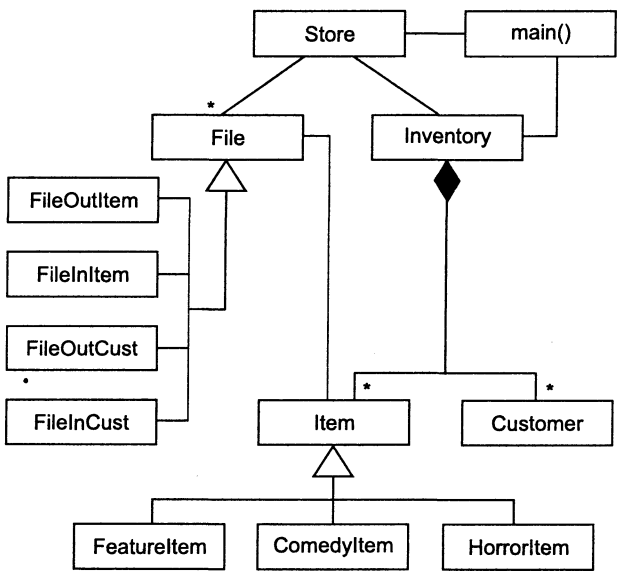


Рис. 14.14. *Диаграмма классов для программ в листингах 14.6–14.16*

Некоторые программисты утверждают, что если объект File открывается для записи данных компонента, он обладает ограниченными возможностями и пытается считывать данные из этого файла или записывать их. Подобные ошибки не должны возникать. Но они появляются. Под давлением каких-либо обстоятельств многие программисты становятся менее внимательными. Совсем неразумно отрицать реальность и настаивать, что если программист квалифицирован и внимателен, то ошибок не будет. Советуем вам избегать ситуаций, которые способствуют появлению ошибок.

На рис. 14.14 представлена диаграмма UML для учебного примера. Классы Item и File используются здесь как базовые классы для специализированных производных классов.

Обратите внимание, что такая структура вовсе не пропагандируется. Просто стоит ознакомиться с этим типом использования наследования. Факторы, которые следует принять во внимание, рассматривая компромиссные варианты,— это количество классов, которые следует реализовать, защита от неверного использования объектов и предотвращение распространения общей информации между классами в приложении.

Итоги

В данной главе наследование сравнивалось с другими методами программирования, например агрегацией и общими связями между классами.

Подчеркивалась ценность других альтернативных методов, поскольку в целом наследование используется слишком часто. Конечно, при использовании наследования работа проектировщиков серверных классов упрощается. Формально задача проектировщика клиентской части не намного труднее. Но это касается только написания программы, а это лишь небольшая часть всего объема работ при реализации программы. Использование наследования заставляет программиста клиентской части больше, чем необходимо, знать о структуре сервера, особенно если иерархия большая по размеру и разветвленная.

Для иллюстрации структур рассмотрено также использование диаграмм UML на примерах. Диаграммы полезны, поскольку позволяют проектировщикам видеть всю картину в целом при обсуждении связей между классами. В примерах использовались только основные конструкции UML. Язык UML в целом очень сложен. Вопрос о том, нужно ли энергично взяться за изучение UML или следует вначале сосредоточиться на глубоком освоении C++, достаточно спорный.

Поскольку эта книга посвящена языку C++, а не объектно-ориентированному анализу и проектированию, навыкам в C++ отдается большее предпочтение. Именно от вашего умения написать на C++ программу, которая передает обязанности серверным классам, зависит качество и удобство сопровождения ПО.

Часть IV

Расширенное использование C++

В последней части книги обсуждается расширенное использование языка C++: виртуальные функции, абстрактные классы, расширенные перегруженные операции, шаблоны, исключительные ситуации, специальные типы и идентификация информации времени выполнения.

В главе 15 "Виртуальные функции и прочее расширенное использование наследования" описывается реализация полиморфизма с виртуальными функциями. Рассматриваются безопасные и опасные преобразования типов между связанными и несвязанными классами, обработка неоднородных списков объектов, принадлежащих различным (но связанным) классам. Затем представлен синтаксис виртуальных функций и показано упрощение клиентской программы, обеспечиваемое этими виртуальными функциями.

Кроме того, рассматриваются чистые виртуальные функции, абстрактные классы и множественное наследование. Хотя виртуальные функции очень полезны для обработки неоднородных списков, важность этой задачи часто преувеличена. Это утверждение справедливо и в отношении множественного наследования — с точки зрения разработки программного обеспечения оно является не столько полезным, сколько сложным.

В главе 16 "Расширенное использование перегрузки операций" обсуждается расширенное использование перегрузки операций: унарные операторы, операторы, возвращающие элемент массива по индексу, операторы вызова функции и операторы ввода/вывода. Как и в других случаях использования перегруженных операторов, эти операторы создают великолепный синтаксис в клиентской программе. Во всем остальном вклад синтаксиса оператора в качество программ на языке C++ ограничен.

В главе 17 "Шаблоны: еще один инструмент проектирования" представлен один из методов языка C++ для проектирования повторного использования: обобщенные шаблоны. Синтаксис определений шаблонов достаточно сложный. Их влияние на размер объектной программы и на ее выполнение часто является вредным. Начинающие программисты на C++ должны соблюдать ограничения при построении своих собственных шаблонных классов.

Однако шаблонные классы, предоставляемые библиотекой стандартных шаблонов C++ (Standard Template Library — STL), спроектированы очень хорошо и должны использоваться, если возможно, для сложных структур данных. Такие классы библиотеки шаблонов представляют исключительный пример проектирования и повторного использования программ.

В главе 18 "Программирование исключительных ситуаций" рассматривается обработка исключительных ситуаций — еще одного метода C++. Это очень интересная область программирования. Возможно, следует попробовать использовать исключительные ситуации в ограниченных размерах. Скорее всего, потом вы сможете оценить, насколько полезна подобная методика в конкретной ситуации. Здесь также обсуждаются специальные типы и идентификация объектов времени выполнения.

Глава 19 "Подведение итогов" является обзором. В ней изложено то, о чем обычно говорится во введении. Мы надеемся, что читатель заинтересуется использованием этого замечательного языка и сможет продуктивно его приме-

Глава 15

Виртуальные функции и прочее расширенное использование наследования

Темы данной главы

- ✔ Преобразования между несвязанными классами
- ✔ Преобразования между классами, связанными наследованием
- ✔ Виртуальные функции: еще одна новая идея
- ✔ Множественное наследование: несколько базовых классов
- ✔ Итоги

В предыдущей главе обсуждалась нотация UML для представления связей "клиент-сервер" и рассматривались методы реализации этих связей в программах на C++.

Наиболее общей является связь включения (композиция или агрегация). В обычной реализации этой связи объект серверного класса становится элементом данных клиентского класса. Объект-сервер обрабатывается исключительно его объектом-клиентом и не используется совместно с другими объектами-клиентами.

Чаще используется связь ассоциация. Если клиентский класс содержит указатель или ссылку на объект серверного класса, он реализует общую ассоциацию между классами. Объект-сервер может совместно использоваться другими клиентскими объектами.

Если у объекта-сервера имеется лишь один клиент, причем только он использует объект-сервер, то последний может быть реализован как элемент данных клиентского класса даже в том случае, если объекты связываются общей ассоциацией, а не агрегацией.

Реализация связи "клиент-сервер" с объектом-сервером как элементом данных объекта-клиента, дает в результате среднюю степень видимости. Объект-сервер видят все функции-члены клиентского класса, но не другие классы в программе. Проектировщикам других классов не нужно изучать использование этого объекта и координировать процесс с другими проектировщиками.

Ограниченная степень видимости достигается, когда объект-сервер реализуется как локальная переменная в функции-члене клиентского класса. В этом случае объект-сервер является видимым только для этой функции-члена, но не для других функций-членов этого или любого другого класса, вне этой функции-члена.

Широкая степень видимости может достигаться за счет того, что объект-сервер передается в функцию-член клиентского класса как параметр. Тогда объект-сервер может ассоциироваться со множеством других объектов вне данного клиентского класса, и эти объекты должны взаимодействовать в использовании объекта-сервера.

Реализация ассоциации путем определения объекта-сервера как локальной переменной в серверном методе приводит в результате к уменьшению зависимостей между частями программы. Программистам, реализующим проект, и лицам, осуществляющим сопровождение, будет проще работать с этой программой. Реализация ассоциации путем передачи объекта-сервера как параметра методу клиента приводит к большей гибкости. Однако проект становится более сложным для программистов и сопровождающих.

Выберите подходящий вариант — наименьшую степень видимости, при которой все еще поддерживаются клиентские требования. Программист C++ должен помнить о выборе одной из трех альтернатив реализации ассоциации. Нотация UML в проекте одна и та же для этих трех методов. Часто проектировщики не знают, какой метод лучше подходит в том или ином случае. Они просто объявляют, что объекты связаны между собой. Следовательно, правильный выбор должен сделать программист C++.

Мы также рассматривали реализацию связей специализации/обобщения между классами. Использование наследования для реализации этой связи между классами позволяет программисту поэтапно строить серверный класс, реализуя часть функциональных возможностей серверного класса в базовом классе, а часть в производном классе. Наследование является мощным, гибким механизмом повторного использования проектов на C++.

В этой главе обсуждается расширенное использование наследования с виртуальными функциями и абстрактными классами. Цель применения наследования — облегчить труд программисту клиентской части за счет выбора оптимальной структуры клиентской программы.

"Подобные объекты" означают, что они имеют атрибуты и операции. Однако некоторые из них отличаются в разных видах объектов. "Схожая обработка" показывает, что клиентская программа интерпретирует эти разные виды объектов в основном одинаковым образом. Однако, в зависимости от типа объекта, некоторые моменты должны быть реализованы по-разному.

Например, в учебном примере предыдущей главы описания фильмов различного типа (художественные, комедии или фильмы ужасов), внесенные в перечень, интерпретировались клиентской программой одинаково. Они считывались из файла, связывались с заказчиками, взявшими их напрокат, подвергались проверке при операциях выдачи и возврата и сохранялись в файле. На нескольких этапах обработки фильмы из разных областей интерпретировались по-разному. Например, при выводе данных о фильме на экран должны отображаться разные метки в зависимости от того, является ли он художественным, комедийным и т. д.

Именно поэтому клиентская программа должна использовать операторы выбора для определения, к какой области относится требуемый фильм и какая обработка должна использоваться. Очевидно, что применение виртуальных функций и абстрактных классов помогает рационализировать клиентскую программу и исключить этот вид динамического анализа из клиентской исходной программы.

Рассмотрим вопросы, связанные с использованием объектов одного класса, в котором предполагается наличие объектов другого класса. Правила языка C++ для такой подстановки с использованием наследования отличаются от правил для несвязанных объектов. Мы попытаемся объяснить вам, в каком направлении следует развивать свою интуицию.

В конце главы показано, как методы использования наследования, виртуальных функций и абстрактных классов могут расширяться в случае, если производный класс содержит более одного базового класса.

Программирование с использованием виртуальных функций и абстрактных классов часто представляют как суть объектно-ориентированного программирования. С практической точки зрения это не так. Большинство программ на C++ имеет дело с взаимодействующими объектами и для них не требуется использование виртуальных функций. Как правило, программы на C++ пишутся (и должны разрабатываться) без использования наследования. Однако программировать с виртуальными функциями полезно. Это один из наиболее сложных вопросов в C++, и хотелось бы надеяться, что вы научитесь использовать виртуальные функции правильно и с удовольствием будете их применять.

Преобразования несвязанных классов

Как утверждалось ранее, C++ стремится поддерживать концепцию строгого контроля за типами. Желательно, чтобы этот принцип современного программирования был воспринят как естественный и привлекательный. Если в соответствии с контекстом программы ожидается объект конкретного типа, то было бы синтаксически неверно использовать вместо него объект другого типа.

Это правило применимо в следующих контекстах:

- Выражения и присваивания
- Параметры функций (включая указатели и ссылки)
- Объекты, используемые как цели сообщений

Два различных класса называются несвязанными, если ни один из них не является прямым или косвенным базовым классом для другого класса. Обратите внимание, что классы, которые не связаны друг с другом через наследование, могут взаимодействовать через агрегации и общие ассоциации. Это прекрасно, но вы еще не можете использовать один класс вместо объектов другого класса. Если эти классы связаны наследованием, это совсем иной случай.

Приведем небольшой пример, демонстрирующий все три контекста, в которых C++ поддерживает строгий контроль типов. Существуют два класса (`Base` и `Other`), которые не связаны наследованием. Функция-член `Base::set()` предполагает параметр целого типа, функция-член `Other::setOther()` — параметр типа `Base`, а функция-член `Other::getOther()` — указатель на объект `Base`. Для простоты не были включены примеры с параметрами-ссылками, но Все, что будет сказано об указателях, также относится и к ссылкам.

```
class Base {                          // первый класс
  int x;
public:
  void set(int a)                     // модификатор
  { x = a; }
  int show() const                    // средство доступа
  { return x; }
};

class Other {                         // второй класс
  int z ;
public:
  void setOther(Base b)               // изменение цели
  { z = b.show(); }
  void getOther(Base *b) const        // изменение параметра
  { b->set(z); }
};
```

В следующей клиентской функции main() определяются три объекта для манипулирования, по одному типа Base и типа Other и один числового типа. Во второй строке представлен параметр правильного типа и верная цель сообщения. Третья строка также правильная и тривиальная. Операнды выражения совместимы между собой, а цель соответствует типу значения.

В данном случае совместимость означает, что для значений двух разных типов (здесь целое и плавающее с двойной точностью) определены одинаковые операции (сложение, присвоение), а значения можно преобразовать из одного типа в другой (целое в значение с двойной точностью) и обратно (с двойной точностью в целое).

Следующие два оператора также правильные. Имена сообщений в них (setOther() и getOther()) совпадают с именами функций-членов, описанных в целевом классе (класс Other), а параметры сообщений имеют правильный тип (класс в четвертой строке или класс Base в пятой строке). Все другие операторы в клиентской программе неправильные и превращены в комментарии. Рассмотрим каждый оператор.

```
int main()
{ Other a; Base b; int x;          // создание объектов
  b.set(10) ;          // OK: правильные типы параметра и цели
  x = 5 + 7.0;         // OK: правильный тип для выражений и lvalue
  a.setOther(b) ;      // OK: правильный тип цели, параметра
  a.getOther(&b) ;     // OK: правильный тип цели, параметра
// b = 5 + 7;          // не OK: не определен operator = (int)
// x = b + 7;          // не OK: отсутствует оператор или преобразование для int
// b.set(a);           // не OK: объект как числовой параметр
// a.setOther(5);      // не OK: невозможно преобразовать число в объект
// a.getOther(&a);     // не OK: отсутствует преобразование Other* в Base*
// b.getOther(&b);     // не OK: неверный тип цели, не элемент
// x.getOther(&b);     // не OK: число в качестве цели сообщения
  return 0; }
```

В первом присваивании (см. ниже) компилятор ожидает значение числового типа. Вместо него используется тип, определенный программистом. Компилятор хотел бы, чтобы было определено присваивание operator=(int) с целыми параметрами типа Base. В этом случае первый оператор стал бы законным. Во втором случае добавлен объект типа, определенного пользователем, и числовая переменная. Такие типы несовместимы. Чтобы это выражение было допустимым, компилятор хотел бы, чтобы был определен operator+(int) для типа Base. В обоих случаях позиция C++ — бескомпромиссная. Строгое соблюдение типа предотвращает появление ошибок на этапе компиляции, а не этапе выполнения.

```
B = 5 + 7; x = b + 7;              // синтаксическая ошибка
```

Следующие два оператора осуществляют передачу параметра. Если функция, например, Base::set(int), ожидает параметр числового типа, то не допускается использование вместо него объекта класса, определенного программистом. Здесь мог бы помочь оператор преобразования, но он рассматривается только в следующей главе. Напротив, если функция, например Other::setOther(Base), ожидает параметр конкретного типа, определенного пользователем, то невозможно вместо него использовать числовое значение или значение некоторого другого типа, определенного пользователем. Во всех этих случаях компилятор отказывается выполнять преобразование значения одного типа в значение другого типа и помечает их как ошибки на этапе компиляции.

```
b.set(a); a.setOther(5);          // синтаксическая ошибка
```

Язык C++ также пытается поддерживать принцип строгого контроля типов для указателей и ссылок. Указатели и ссылки объединены вместе, поскольку правила для них одинаковы. Если у функции есть параметр, который определяется

как указатель (или ссылка) для объекта того же самого типа, то передача ему указателя (или ссылки) для объекта любого другого типа, встроенного или определенного программистом, будет ошибкой.

```
a.getOther(&a) ;            // синтаксическая ошибка
```

Вы не можете вызвать функцию, которая ожидает указатель, с ссылкой или с объектом как фактическим параметром, даже если ссылка и объект представляют собой один тип. Подобным образом, если функция ожидает параметр-ссылку, то фактический параметр не может быть указателем, даже если он имеет тот же тип.

Для целей сообщений концепция сильного контроля типов проявляется в ограничении набора сообщений. Если имя сообщения, отправленного объекту, не обнаруживается в спецификациях класса, возникает ошибка независимо от того, можно найти эту функцию в любом другом классе или нет. Для компилятора достаточно, что сообщение не находится в классе, к которому принадлежит цель сообщения. И, конечно, невозможно отправить сообщение числовой переменной или значению, потому что они не принадлежат ни одному классу и не могут соответствовать никаким сообщениям. Компилятору требуется переменная с типом, определенным пользователем, в левой части оператора выбора точки (dot selector operator).

```
b.setOther(b);  x.setOther(b) ;          // неверные типы цели
```

Переменная-указатель (и ссылка) может указывать только на значение типа, использованного при его объявлении. Это еще одно проявление строгого контроля типов. В приведенном ниже фрагменте программы вторая строка является правильной, а третья строка неправильной.

```
Other a; Base b;
Base &r1 = b; Base *p1 = &b;            // OK: совместимые типы
Base &r2 = a; Base *p2 = &a;            // несовместимые типы
```

Строгий и слабый контроль типов

Это идеальное решение всех проблем. Однако в языке C++ в строгих правилах допускается множество исключений.

Например, все числовые типы рассматриваются эквивалентными с точки зрения проверки типа. Их можно свободно смешивать в выражении, и компилятор будет преобразовывать "младшие" операнды в "старшие" операнды так, чтобы все операторы применялись к операндам одного типа. В правой части присваивания или в параметре вызова функции, когда предполагается значение "младшего" числового типа, можно использовать значение "старшего" числового типа. Компилятор снова "молча" преобразует "старшее" значение (например, длинное целое) в "младшее" значение (например, символ). Компилятор предполагает, что вы сами знаете, что делаете.

Некоторые компиляторы при использовании числового значения "старшего" типа, когда предполагается значение "младшего" типа, могут выдавать предупреждающее сообщение. Это происходит, например, при попытке сжать значение плавающего типа с двойной точностью в целое или в символьную переменную. Однако компилятор выдает лишь предупреждение, а не показывает синтаксическую ошибку. Вслед за C, C++ позволяет использовать явное приведение, чтобы показать устройству считывания желание проектировщика преобразовать значение одного числового типа в значение другого числового типа.

Однако эту опцию может использовать только опытный программист. Для программистов, стремящихся к краткости, снова вслед за C, C++ признает законность всех неявных преобразований. Нестрогое отношение к потенциальной потери точности применяется как к присваиванию, так и к передаче параметров.

С этой точки зрения, C++ (подобно C) является языком со слабым контролем типов. Здесь компилятор предполагает, что вам известно, что вы делаете. Если вам неизвестно или вы не уделяете внимания этой стороне вычислений, будем считать, что ваши вычисления действительно не зависят от точности усеченных значений.

C++ также поддерживает другие исключения из правил строгого соблюдения типов, которые несовместимы с языком C. Эти исключения вытекают из использования специальных функций-членов и приведения типов:

- Конструкторы преобразования
- Операторы преобразования
- Приведение указателей (или ссылок)

Эти специальные функции представляют компилятору C++ способы указания принять клиентскую программу, которая нарушает правила строгого контроля типов.

Конструкторы преобразования

Предположим, что класс Base предоставляет конструктор преобразования с числовым параметром.

```
Base::Base(int xInit = 0)       // конструктор преобразования
{ x = xInit; }
```

При наличии этого конструктора оператор компилируется.

```
a.setOther(5);                  // неверный тип, но не синтаксическая ошибка
```

Компилятор интерпретирует подобное сообщение так:

```
a.setOther(Base(5));            //с точки зрения компилятора
```

Создается временный объект класса Base, инициализированный вызовом конструктора преобразований. Он используется как фактический параметр правильного типа, а затем уничтожается. Следовательно, требования сильного контроля типов рассматриваются на уровне компилятора — функция получает значение нужного типа. Эти требования не удовлетворяются на уровне программиста. Программист передает в setOther() параметр неверного типа.

Обратите внимание на то, что конструктор получает значение параметра по умолчанию. Для чего это сделано? Прежде чем этот конструктор был добавлен к классу Base, в нем имелся конструктор по умолчанию, предоставляемый системой. Вы можете определить объекты класса Base без параметров. При наличии "на месте" конструктора преобразования система убирает конструктор по умолчанию, а определения объектов Base без параметров не становятся синтаксическими ошибками.

Как упоминалось ранее, C++ может превратить существующую программу в синтаксически неверную, когда новый сегмент программы (в данном случае конструктор) добавляется без удаления чего-либо из программы. В других языках добавление новой программы способствует правильному выполнению несвязанных частей программы, однако нельзя всю программу сделать синтаксически неверной. С одной стороны, это печально, потому что добавление несвязанной программы не должно вызывать проблем в существующих частях программы. С другой стороны, компилятор уведомляет программиста о проблемах во время компиляции, а не при выполнении программы.

Чтобы избежать подобных трудностей, можно было бы добавить к классу Base определенный программистом конструктор, задаваемый по умолчанию, который ничего не делает. В этом случае не требуется инициализация объекта в конкретном

значение, которое в дальнейшем даже не используется. Но из-за небрежности было добавлено нулевое значение параметра по умолчанию, чтобы обеспечить компилирование существующей клиентской программы. В чем недостаток подобного решения? Предположим, что нулевое значение используется где-то еще. Между тем оно не применяется. Мы знаем, что оно не используется, но программист, осуществляющий сопровождение программы, должен будет это установить. Следовательно, усложняется чтение программы.

Итак, добавление конструктора преобразования к классу Base позволяет скомпилировать вызов функции-члена Other::setOther(Base) с реальным параметром числового типа.

```
a.setOther(5);                    // то же, что и a.setOther(Base(5));
```

Когда компилятор не находит точного соответствия для типа параметра, он осуществляет поиск варианта числового преобразования. Если соответствующее числовое преобразование отсутствует, то осуществляется поиск объединенного, числового и определенного программистом, преобразования. Конструктор преобразования является одним из возможных преобразований, определенных программистом.

При наличии этого конструктора следующий оператор также становится допустимым, потому что компилятор вызывает конструктор преобразования, отвечая требованиям строгого контроля типов.

```
b = 5 + 7;                        // не ошибка: то же, что и b = Base(5+7);
```

Компилятор пытается предугадать цели программиста. Одна из задач проектирования на C++ — избежать этого и позволить программисту явно указать назначение данной программы. Один из способов явного указания, что подразумевается, состоит в использовании явного приведения типов. Однако, согласно правилам слабого контроля типов в C/C++, для преобразования числовых типов явное приведение не требуется, а вызов конструкторов преобразований может выполняться без явного вызова. Что делать? Стандарт ISO/ANSI предлагает компромисс. Если проектировщик класса чувствует, что конструктор преобразования должен вызываться только явно, зарезервированное слово explicit используется как модификатор (см. главу 10).

```
explicit Base::Base(int xInit = 0)          // отсутствуют неявные вызовы
{ x = xInit; }
```

Использование ключевого слова explicit является необязательным. Если воспользоваться им при проектировании класса Base, то написать программу будет трудно. Также сложнее будет написать клиентскую программу (программист клиентской части должен будет использовать явное приведение типов), но понять полученную в результате программу станет легче. Если это ключевое слово не использовать, ухудшится но качество программы. Прийти к компромиссу сложно.

C++ допускает все виды скрытого преобразования, но зарезервированное слово explicit не разрешает их использовать. В настоящее время оператор снова вызывает синтаксическую ошибку, несмотря на присутствие конструктора преобразования, он требует явного приведения типов.

```
a.getOther(5) ;              // недопустимо, если конструктор определяется явно
```

Обратите внимание, что неявные преобразования применяются только к параметрам, которые передаются по значению. Это не нужно в отношении параметров ссылок и указателей. Добавление конструктора преобразования не вызывает компиляции вызова Other::getOther(Base* b) с цифровым параметром.

```
int x = 5;
a.getOther(&x) ;             // это все еще синтаксическая ошибка
```

Приведение указателей

Правила неявных преобразований (слабый контроль за типами для значений) применяются только к значениям, а не к ссылкам или к указателям (строгий контроль за типами для значений). Однако явные преобразования могут использоваться для параметров любого характера. Можно ли передать указатель целого типа для указателя Base? Нет, в соответствии с правилами строгого контроля типов, следующая строка ошибочна:

```
a.getOther(&x);                    // синтаксическая ошибка
```

Однако всегда можно указать компилятору, что он не должен принимать этот код. Для этого используется явное приведение для правильного типа.

```
a.getOther((Base*)&x);             // не проблема, преобразование OK!!
```

В этом вызове функции создается и инициализируется указатель Base. Он обозначает ячейку памяти, содержащую x. Внутри getOther() сообщения класса Base направляются в область, занимаемую x. Поскольку методы Base не знают о структуре данных x, они легко могут ее повредить. Вся операция в целом вообще не имеет смысла, но в C++ она законна. Если утверждается, что так нужно, компилятор не будет спорить с вами.

Это же справедливо в отношении преобразования указателей (или ссылок) для любого типа указателей или ссылок. Неявные преобразования разных типов не допускаются. Например, ошибкой является следующее:

```
a.getOther(&a);        // ошибка: отсутствует преобразование из Other* в Base*
```

Метод getOther() ожидает указатель типа Base. Вместо этого он получает объект типа Other. В соответствии с принципом строгого контроля типов компилятор помечает эту строку как синтаксическую ошибку — неявное приведение указателей (или ссылок) разного типа не допускается. Однако компилятор разрешает вызов функции с явным оператором выбора.

```
a.getOther((Base*)&a);             // не проблема, явное преобразование - OK
```

При этом указатель на объект класса Base создается и инициализируется как указатель на объект a в Other. Данный указатель передается методу getOther() как фактический параметр. Внутри метода getOther() этот указатель используется для передачи объекту Other сообщений, принадлежащих классу Base. Компилятор не может пометить их как ошибочные. Выполнение программы приводит к аварийному завершению или дает неверные результаты. Эта программа бессмысленна, но в C++ она допустима.

Операторы преобразования

Операторы преобразования используются в C++ как обычные приведения типов. Когда они применяются к объектам типов, определенных пользователем, они обычно возвращают значение одного из компонентов объекта. Например, приведение к int, примененное к объекту типа Other, может вернуть значение элемента данных x. Использование этого оператора исключает синтаксические ошибки при применении объекта типа Other, в котором предполагается целый тип (или другой числовой тип).

```
b.set(a);                  // то же, что и b.set(int(a));
```

Это пример клиентской программы, которая поддерживается за счет добавления соответствующих сервисов для серверного класса Other. Способы реализации данного вида сервиса будут изложены в главе 16 "Расширенное использование перегрузки операций". Однако решение использовать операторы преобразования — еще один удар по системе строгого контроля типов в языке C++.

Если эта программа была написана, чтобы выполнить преобразование из Other в int, прекрасно. (Лучше было бы использовать явное приведение типов.) Если по ошибке объект a использовался вместо целого, компилятор не сообщит вам об этом. Защита строгого контроля типов удалена, а ошибка обнаруживается во время рабочего тестирования и отладки.

C++ является языком со слабым контролем числовых типов. Можно свободно выполнять преобразование из одного числового типа в другой и явное приведение типов не требуется. Будьте осторожны, чтобы не сделать ошибку.

В C++ ведется строгий контроль типов, определенных программистом. Язык не обеспечивает приведение типов для числовых типов и типов, определенных пользователем, или разных типов, заданных пользователем. Ошибка помечается как синтаксическая, и ее можно скорректировать до выполнения программы.

Конструкторы и операторы преобразований ослабляют систему строгого контроля типов в C++ для типов, определенных пользователем. Они допускают явные и неявные преобразования цифровых типов и типов, определенных программистом, но сделанная при этом ошибка не помечается как синтаксическая.

В той степени, в которой это касается указателей (или ссылок), C++ обеспечивает смесь сильного и слабого контроля типов. Указатели не обозначают объекты с типами, отличающимися от их собственного типа. Однако они могут свободно преобразовываться в указатели любого другого типа. Следует использовать явное приведение типов (в отличие от числовых значений неявные приведения не допускаются, даже для указателей на числовые типы). Важно, чтобы память, на которую указывают указатели (или ссылки), использовалась после этого приведения правильно.

Преобразование классов, связанных наследованием

Использование наследования вводит дополнительные возможности для применения объекта одного типа, когда предполагается объект другого типа. Классы, связанные общедоступным наследованием, не являются полностью несовместимыми, поскольку объект производного класса содержит все операции и элементы набора данных, которые имеются у объекта базового класса. Вы можете назначить объект одного класса для объекта другого типа (возможно, используя явное приведение типов). Вы можете передать объект одного класса как параметр, когда ожидается параметр другого класса.

Правила C++ для преобразования классов, связанных наследованием, не очень сложны. Кажется, однако, что они выполняются вопреки широко распространенной интуиции программирования. Если это так, то постарайтесь направить интуицию в нужном направлении.

Помните: когда производный класс открыто наследуется из базового класса, C++ поддерживает явные стандартные преобразования из производных объектов в открытые базовые классы. Кроме того, допускаются преобразования из базового объекта в производный класс, но для них потребуется явное приведение. Это правило применяется к объектам классов, ссылкам на объекты классов и указателям на объекты.

Чтобы это правило стало понятным, рассмотрим несколько примеров и представим диаграммы, иллюстрирующие преобразования из одного типа в другой.

Важными являются концепции безопасных и опасных преобразований.

Безопасные и опасные преобразования

Рассмотрим фрагменты программы, в которых используются числовые переменные и которые показывают обработку переменных разных типов.

```
int b = 10; double d;
d = b;                  // из "меньшего" в "больший" тип: безопасная пересылка
```

В этом примере небольшой объем данных (4 байта на машине автора) пересылается в большую часть данных (8 байт на машине автора). Какие бы значения не содержала целая переменная, она может быть сохранена в переменной с плавающей точкой двойной точности. При пересылке она не теряет ни точность, ни значение. Именно поэтому это преобразование считается безопасным, и компилятор C++ не выдает никаких предупреждений для программы такого типа.

Рассмотрим движение данных в обратном направлении.

```
int b; double d = 3.14;
b = d;                    // из "большего" в "меньший" тип : опасная пересылка
```

В данном случае значение в 8 байт с плавающей точкой двойной длины может не поместиться в меньшей области, выделяемой для переменной целого типа. Дробная часть будет потеряна. Если значение двойной длины находится вне допустимого диапазона для целых чисел, то также будет потеряно и само значение. Поэтому такое преобразование является опасным, и для подобных программ компиляторы C++ могут выдавать предупреждения.

Однако C++ не считает такое присваивание незаконным. Прежде всего, не все опасные операции являются неверными. Значение двойной длины может быть небольшим, поэтому его можно легко сохранить в значении целого типа. В рассматриваемый момент у значения двойной длины может отсутствовать дробная часть.

Только программист может оценить язык C++. Если он знает, что делает (какое значение преобразуется и что случится с ним в результате пересылки), и доволен полученными результатами, прекрасно. Если нет, то C++ не будет выступать в роли "старшего брата".

Обсудим преобразование переменных разных классов. В отличие от предыдущей части, в которой обсуждалось преобразование объектов несвязанных классов, предположим, что классы связаны наследованием.

Рассмотрим классы Base (содержит один целый элемент данных с размером в 4 байта) и Derived (включает два элемента данных целого типа с размером в 8 байт)

```
class Base {                                    // класс Base
protected:
  int x;                                        // защищенные данные
public:
  Base(int a)                                   // используется в Derived
  { x = a; }
  void set (int a)                              // наследуется
  ( x = a; }
  int show () const                             // наследуется
  { return x; }
} ;

class Derived : public Base {
  int y;                                        // в дополнение к x
public:
  Derived (int a, int b) : Base(a), y(b)        // список инициализации
    { }                                         // пустое тело
  void access (int &a, int &b) const            // дополнительная возможность
  { a = Base::x; b = y; }                       // извлечение данных объекта
} ;
```

Применим эту логику "соответствия размерам" к передаче данных между переменными двух классов.

```
Base b(30); Derived d(20,40);
d = b;                          // из "меньшего" в "больший" тип: соответствует
```

Подобно предыдущему примеру с числовыми размерами, передадим "меньшее" значение (4 байта) большему значению (8 байт). У объекта назначения места для осуществления перемещения с избытком, причем данные не будут потеряны. Теперь переместим данные в противоположном направлении.

```
Base b(30); Derived d(20,40);
b = d;                          // из "большего" в "меньший" тип: не помещается
```

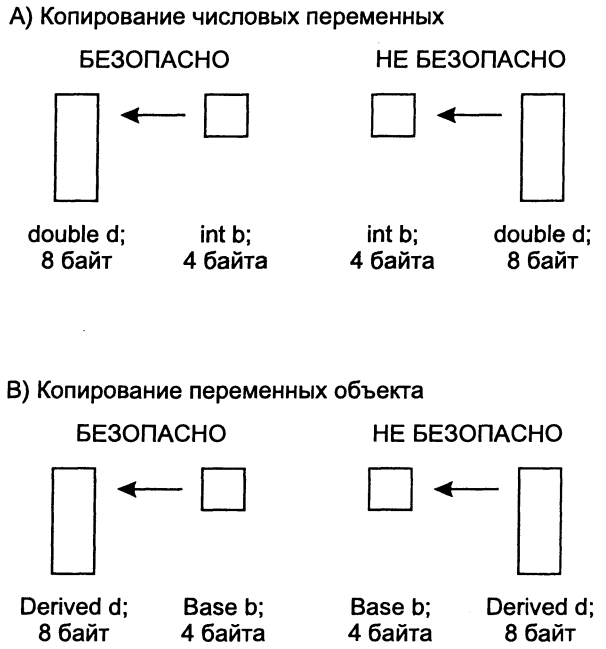


Рис. 15.1. *Пересылка данных между значениями различных размеров: неверный вариант*

Дополнительные данные будут отброшены, поскольку они совершенно не нужны объекту Base. Объект Base всегда будет в согласованном состоянии. Это безопасно.

Когда данные перемещаются от объекта Base к объекту Derived, объект Base располагает достаточными данными для размещения части Base объекта Derived. Данные для установки части Derived объекта Derived неоткуда взять, и это проблема. Объект Derived переходит в несогласованное состояние.

Это опасно, и C++ выдает синтаксическую ошибку. На рис. 15.2 показано, что для числовых значений важно сохранить значения и точность, а для значений класса данные должны быть доступными для установки всех сохраняемых значений объекта назначения.

Поможет ли в этом случае явное приведение типов? Прежде всего, язык C++ предоставляет средства для указания компилятору, что вам известно о своих действиях.

```
Base b(30); Derived d(20,40);
d = (Derived)b;          // данные взять неоткуда
```

В данном случае перемещается большее значение Derived в переменную меньшей длины Base. Памяти, выделенной для переменной Base, недостаточно для вмещения всех элементов данных значения Derived. Большое значение не помещается в меньшую область. На рис. 15.1 показано перемещение данных из меньшего значения в большее, и оно помечается как безопасное. Также представлено перемещение данных из большего значения в меньшее, которое помечено как опасное.

Эта логика хорошо работала для числовых переменных, но не применима для объектов классов, связанных наследованием. Необходимо как можно быстрее развить интуицию. Реальным вопросом для объектов класса является не наличие достаточного пространства, а доступность данных для согласованного состояния объекта.

При перемещении данных от объекта Derived в объект Base объект Derived обладает достаточными данными для заполнения объекта Base.

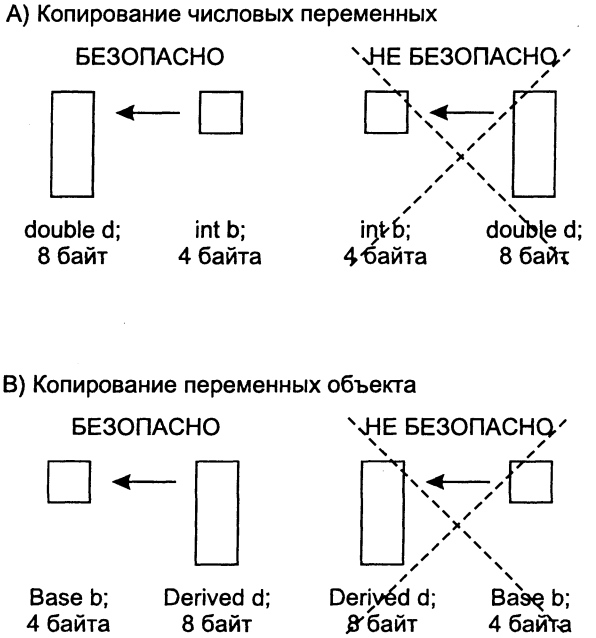


Рис. 15.2. *Пересылка данных между значениями различных размеров: правильный вариант*

Это все еще синтаксическая ошибка, потому что объект Base не может предоставить отсутствующие данные. Возможно, вам захочется использовать нули для установки элементов данных класса Derived, которые не могут быть инициализированы из объекта Base (здесь это элемент данных y). Однако это невозможно выполнить по умолчанию: компилятору не известно, допустимы ли нули. Чтобы уведомить об этом компилятор, можно перегрузить оператор присваивания класса Derived. Передайте ему параметр Base, скопируйте поля параметра и установите остальные поля в любое значение.

```
void Derived::operator = (const Base &b)          // параметр Base
  { Base:: x = b.show(); y = 0; }                 // компромисс
```

Другие примеры операторов присваивания классов можно найти в главе 11 Там операторы присваивания включают параметр либо того же класса, что и класс назначения присваивания, либо тип одного из компонентов класса. Здесь представлен оператор присваивания, параметр которого относится к классу Base. Это замечательно, поскольку перегруженный оператор присваивания представляет собой просто функцию C++, и можно спроектировать функции C++ с помощью параметров любого типа. Объект базового класса является одним из компонентов объекта производного класса.

Почему тело оператора присваивания настолько сложно? Почему необходимо относиться к классу Base очень внимательно? Почему используется операция явного задания объекта, а также функция show()? Прежде всего, элемент данных x защищен в классе Base, не правда ли? Возникает много вопросов. Главный из них: можно ли упростить оператор присваивания и записать его следующим образом?

```
void Derived::operator = (const Base &b)        // параметр Base parameter
  { x = b.x; y = 0; }                           // отлично!!
```

Ответ на этот вопрос будет дан позже в этой главе.

Второй вопрос: в программе имеются две строки, которые пытаются скопировать объект класса Base в объект класса Derived. Какую строку поддерживает этот оператор назначения?

```
d = b;                          // то же, что и d.operators (b); ??
d = (Derived)b;                 // то же, что и d.operator=(b); ??
```

Оператор назначения поддерживает первую строку. Вторая строка не вызывает оператор присваивания, поэтому это делается в первой строке. Уже говорилось, что всегда следует помнить о различии между присваиванием и инициализацией, поскольку в языке C++ это разные вещи. Для поддержки второй строки требуется функция-член, которая будет вызываться при компиляции оператора приведения. Что такое приведение типов? Приведение типов означает вызов конструктора. Конструктор принадлежит к классу приведения типов. Следовательно, для поддержки второй строки программы необходимо записать следующее:

```
Derived::Derived(const ??? &b)          // какой тип параметра?
{ /* what do I do here? */ }
```

Теперь имя конструктора известно. Тип данного параметра предполагается использовать для инициализации полей объекта класса Derived. Согласно строке программы параметр должен иметь тип Base.

```
Derived::Derived(const Base &b)         // параметр Base
{ /* what do I do here? */ }
```

Обратите внимание, что часто в обсуждении используются слова "строка программы, которую требуется поддерживать" в различных видах. Причина заключается в следующем: внешний вид серверных классов определяется исходя

из принятого решения о том, что требуется поддержать в клиентской программе. Вам надо скопировать параметр в часть Base объекта класса Derived и как-то изменить часть Derived объекта, что сохранит объект в согласованном состоянии. Подобно оператору присваивания, можно установить элемент данных Derived в ноль.

```
Derived::Derived(const Base &b)          // параметр Base
{ Base::x = b.x; y = 0; }                // Это неправильно!
```

Это неверно. Здесь не был учтен совет, что надо постоянно помнить о процессе построения объекта. Предполагалось, что конструктор вызывается при построении объекта, а не после того.

При построении объекта его компонентам выделяется область памяти. Конструктор для каждого компонента вызывается до того, как следующему компоненту выделяется память. Сначала для объекта класса Derived размещается компонент Base. Следовательно, вызывается конструктор Base. Какой конструктор? Поскольку части Base не передавались никакие данные, это конструктор по умолчанию. Определим класс Base и проверим, содержит ли он конструктор по умолчанию. Нет, он содержит конструктор преобразования без установленного по умолчанию значения параметра. Значит, попытка вызвать спроектированный конструктор приведет к вызову пропущенного конструктора Base, в результате чего появится синтаксическая ошибка. Помните об этих моментах.

Вам необходимо сделать доступным определяемый по умолчанию конструктор Base. Можно добавить конструктор к классу Base или значение параметра по умолчанию к существующему конструктору преобразования Base. Кроме того, рекомендуем воспользоваться списком функции инициализации в проектируемом конструкторе Derived, а не предоставлять конструктор по умолчанию классу Base. Используем список инициализации.

```
Derived::Derived(const Base &b)          // параметр Base
  : Base(b.x)                            // передача элемента данных?
{ y = 0; }                               // как вам это нравится?
```

Но это также неверно. Элемент данных x в классе Base не является общедоступным, следовательно, к нему нет доступа из-за пределов области действия класса. Некоторые программисты доказывают, что код конструктора, который соответствует оператору Derived::scope, находится в классе Derived, а у класса Derived есть доступ к незакрытым элементам данных в Base. Это действительно так. Но объект-параметр b отличается от цели сообщения, и методы класса Derived не могут осуществлять доступ к его не общедоступным данным. Оператор присваивания класса Derived может осуществлять доступ только к своей внутренней организации, определенной в Base. Именно поэтому должна использоваться функция-член Base.

```
Derived::Derived(const Base &b)          // параметр Base
  : Base(b.show())                       // передача возвращаемого значения
{ y = 0; }                               // неужели некрасиво?
```

Это правильно, но, вероятно, слишком совершенно. Вызывается функция-член show() в Base, возвращаемое значение передается в конструктор преобразования Base. Проще вызвать конструктор копирования Base. Он всегда доступен. Поскольку класс Base не осуществляет динамическое управление своей памятью, то не требуется для себя писать специальный конструктор копирования.

```
Derived::Derived(const Base &b): Base(b), y(0)    // копирование
{ }                                               // это действительно красиво
```

Мы рассматриваем здесь копирование объекта Derived в объект Base и копирование объекта Base в объект Derived (операция является ошибкой, если только она не поддерживается конструктором копирования или оператором присваивания). История применяется как к использованию объектов в операторах присваивания, так и к передаче объектов функции по значению.

При изложении этой истории проводилась аналогия с целостностью данных. Было сказано, что копирование объекта Derived в объект Base безопасно, поскольку объект Derived содержит все данные Base (плюс дополнительные), а полученный в результате объект Base будет находиться в согласованном состоянии. Отмечалось, что копирование объекта Base в объект Derived не является безопасным, поскольку объект Base представляет собой небольшой объект, у которого отсутствуют данные, необходимые для инициализации крупных объектов Derived. Следовательно, подобная операция может оставить объект Derived в несогласованном состоянии.

Использование объекта одного типа там, где ожидается объект другого типа, будет безопасным, только если используемый объект может выполнять все операции, запрашиваемые от ожидаемого объекта. Этот подход будет представлен при обсуждении использования указателей (или ссылок) одного типа, когда ожидается указатель (или ссылка) другого типа.

Преобразование указателей и ссылок в объекты

Поговорим об использовании указателей и ссылок на объекты Derived и Base. Обсудим только указатели, но все сказанное о них также применяется и к ссылкам. Будет рассматриваться иерархия только двух классов, Base и Derived, но все это применимо и к другим иерархиям классов, в которых базовый класс содержит другие производные классы, а производные классы используются как база для других классов.

В первую очередь динамически создается объект Base. С помощью указателей Base и Derived предпринимается попытка вызова его методов. Затем динамически создается объект класса Derived и делается попытка вызова его методов с помощью указателей Derived и Base. После этого результаты обобщаются для передачи параметров функциям по указателю и по ссылке.

Пусть объект класса Base помещается в динамически распределяемую область памяти. Используйте указатель Base, который обозначает этот объект для доступа ко всем методам Base (например, show()), но не к методам любого другого класса. Методы класса Derived (например, access()) также недоступны указателю Base просто потому, что они не принадлежат к классу Base.

При обработке этих сообщений компилятор идентифицирует имя указателя (pb), который указывает на неименованный целевой объект, использует объявление указателя для создания класса указателя (Base), переходит к спецификациям класса и осуществляет поиск по спецификациям класса по имени метода. Если имя обнаруживается (в данном случае show()), то генерируется код объекта. Если имя метода с соответствующей сигнатурой не находится (в данном случае access()), это синтаксическая ошибка.

```
int x, y; Base *pb = new Base(60);          // базовый объект
cout << " x = " << pb->show() << endl;      // это OK
pb->access(x,y);                            // это невозможно в любом случае
```

Затем попытаемся установить указатель класса Derived на такой же объект класса Base. Алгоритм разрешения имени функции, описанный ранее, расширяется, когда указатель (или переменная) принадлежит классу Derived. Если метод с именем сообщения обнаруживается в классе Derived, замечательно. В противном случае компилятор переходит к описанию класса Base. Это происходит только тогда, когда метод с соответствующей сигнатурой не находится в описании класса Base, при этом компилятор помечает вызов функции как синтаксическую ошибку.

Следовательно, указатель класса Derived будет в состоянии использовать все элементы определения класса Derived. "Любой элемент определения класса" означает элементы, определенные в классе Derived (например, access()), и наследованные элементы (например, show()). При попытке установить указатель Derived на объект Base компилятор останавливается, даже не переходя к следующим операторам.

```
Derived *pd = pb;                    // указывает на объект Base object: не OK
cout << " x = " << pd->show() << endl;          // это было бы OK
pd->access(x,y);                     // но этого следует избегать
```

Преобразование из объекта Base в объект Derived небезопасно. Следовательно, преобразование из указателя (или ссылки) Base в указатель (или ссылку) Derived не является безопасным. Все они создают синтаксические ошибки.

Однако при формальном объяснении трудно понять, что происходит. Необходимо развивать соответствующую интуицию. К сожалению, программирование на традиционных языках не развивает интуицию в отношении преобразования классов, подстановки объектов и разрешению и запрещению вызовов функций. Попробуйте развить свою интуицию в отношении преобразования указателей и ссылок, для этого используйте графические представления и аналоги, основанные на размере объектов и их способности выполнять работу для клиентской программы.

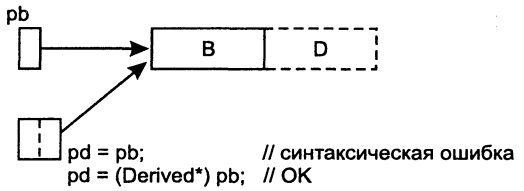


Рис. 15.3. *Доступ к объекту Base через указатель класса Derived: небезопасно*

На рис. 15.3 указатель Derived показан большим прямоугольником в отличие от указателя Base, поскольку указатель Derived может предоставить больше элементов для определения класса, чем Base. Здесь важна именно способность объектов отвечать на сообщения и выполнять работу для клиентской программы.

Указатель Base может осуществлять доступ только к элементам определения классов класса Base, а указатель Derived — к элементам определения классов как класса Base, так и класса Derived. Динамически выделенный объект класса Derived представлен в виде большого прямоугольника не потому, что он занимает больший объем памяти, а потому, что он содержит больше возможностей. Даже если он не имеет дополнительные элементы данных, он всегда содержит дополнительные функции-члены. Объект класса Base показан в виде присоединенного прямоугольника, обозначенного пунктирной линией, для указания возможностей, имеющихся в объекте класса Derived, но отсутствующих в объекте класса Base. Буквы B и D внутри прямоугольников объектов обозначают часть Base и часть Derived объекта соответственно.

В результате копирования содержимого указателя pb в указатель pd они оба указывают на один и тот же объект Base (см. рис. 15.3). Здесь важно, что указатель, который указывает на объект, может вызвать возможности объекта в соответствии с типом указателя, а не с типом объекта.

Предположим, что объекты базового класса являются слабыми, неспособными объектами, которые почти ничего не могут. А производные объекты окажутся большими, сильными, мощными объектами, способными выполнить все.

Думайте об указателях базового класса как о тонких, слабых указателях, которые могут извлекать только возможности, определенные в классе Base. Например, указатель Base может извлечь метод show(), определенный в классе Base, но не метод access(), определенный в классе Derived.

Наконец, указатели типа Derived должны представляться большими, сильными, дальнозоркими указателями, которые могут извлекать множество возможностей (как show() из Base, так и access() из Derived).

Когда устанавливается мощный указатель класса Derived для указания на слабый объект класса Base, он может извлекать намного больше, чем объект может предоставить. Из приведенных выше двух строк, расположенных за назначением указателя pd = pb, первая строка (вызов show()) правильная, а вторая (вызов access()) — нет, в Base отсутствует access(). Именно поэтому C++ объявляет преобразование pd = pb синтаксической ошибкой, чтобы воспрепятствовать таким вызовам, как pd->access(x,y). Сам по себе вызов является синтаксически правильным (pd принадлежит классу Derived), но семантически бессмысленным — объект Base не может ответить на это сообщение.

Конечно, компилятор может видеть то, что уже знаете вы. Он понимает, что указатель pd класса Derived указывает на объект Base, следовательно, вызов pd->show() должен быть доступен, а вызов pd->access(x,y) — не разрешен. Но в этом случае проектировщик компилятора должен уметь делать очень много. В языке C++ часто отдается предпочтение проектировщику компилятора. От компилятора C++ не требуется выполнения анализа потока данных. Вы же должны изучить правила преобразования.

Итак, преобразование pd = pb не является безопасным и помечается как синтаксическая ошибка. Но если у вас были только самые лучшие намерения? Что, если предполагалось только вызвать функции Base (например, show()), а не функции Derived (например, access())? Должен существовать механизм для указания компилятору того, что неизвестно ему, но знакомо вам, а именно, что будут использоваться только возможности класса Base. Такой механизм действительно существует (см. главу 6). Он называется приведение типов.

Используя приведение типов, от компилятора запрашивается выполнение безопасного преобразования. При желании "расправиться без суда" надо вызвать pd->show(), а не pd->access(x,y).

```
Derived *pd = (Derived*)pb;                // Компилятор, мне нужно именно это
cout <<"x="<< pd->show()<<endl;            // сделаем то, что безопасно
// pd->access(x,y);                        // об этом не стоит даже думать!
```

Обратите внимание, что имя приведения типов включает не просто имя класса, но и указатель. Было бы неправильно опустить нотацию указателя. Также не допускается использование функциональной записи. Рекомендуем ее использовать, только когда имя типа является идентификатором, а Derived* не представляет собой идентификатор (вспомните, что звездочка в идентификаторах C++ не допускается).

```
Derived *pd = (Derived)pb;      // невозможно преобразовать указатель в объект
Derived *pd = Derived*(pb);     // недопустимое имя для функционального
                                // приведения типов
```

Если требуется использовать функциональную запись, применяйте typedef для составления имени типа, например DerivedPtr.

```
typedef Derived* DerivedPtr;        // имя нового типа: идентификатор
Derived *pd = DerivedPtr(pb);       // идентификатор: OK для такого приведения
```

Создадим объект класса Derived в динамически распределяемой области памяти. Используя указатель класса Derived (большого, мощного, дальнего действия), можно вызвать возможности, унаследованные из класса Base, так и определенные в классе Derived. И это нормально, поскольку объект класса Derived (большой и мощный) располагает всеми этими возможностями.

```
Derived *pd = new Derived(50,80);          // объект Derived может все
cout <<" x="<< pd->show() << endl;         // OK для вызова базового метода
pd->access(x,y) ;                          // OK для вызова производного метода
```

Скопируем содержание указателя Derived в указатель Base (тонкий, слабый и ближнего действия). Указатель Base может осуществлять доступ только к возможностям объекта Base, на который он ссылается. Попытка осуществить доступ к возможностям класса Derived через этот указатель бесполезна. Здесь эти возможности отсутствуют, и компилятор сгенерирует синтаксическую ошибку.

```
Base *pb = pd;                          // указатель на тот же объект
cout <<" x="<<pb->show()<<endl;         // Метод Base здесь есть
// pb->access(x,y);                     // ошибка: отсутствует в классе Base
```

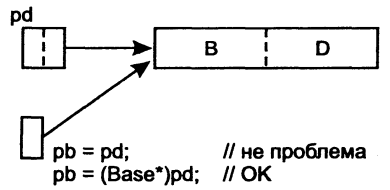


Рис. 15.4. *Доступ к объекту Derived через указатель класса Base: безопасно*

На рис. 15.4 показано, что из этого получается. Снова указатель Derived обозначается в виде большого прямоугольника в отличие от указателя Base. Прямоугольник для обозначения динамически выделенного объекта класса Derived больше прямоугольника для указателя Base. Буквы B и D внутри прямоугольников объектов обозначают часть Base и часть Derived объекта соответственно.

Как представлено на рис. 15.4, в результате копирования содержимого указателя pd в указатель pb они оба указывают на один и тот же объект Derived (большой, сильный, мощный, способный выполнить все, что может и объект Base, плюс намного больше). Однако этот указатель pb класса Base является тонким, слабым и близоруким, он способен извлекать только возможности Base объекта, но не возможности Derived.

Когда определяется, что слабый указатель класса Base ссылается на большой объект класса Derived, этот слабый указатель не может повредить. Он может вызвать только возможности Base (например, show()), а они всегда присутствуют в мощном объекте Derived. Именно поэтому язык C++ воспринимает преобразование pb = pd как безопасное. Также он воспринимает копирование большого объекта класса Derived в маленький объект класса Base. Это преобразование не может привести к пересылке безымянному объекту сообщения, на которое объект не может ответить.

Если быть точным и указать сопровождающему программисту то, что было известно на момент написания программы (указатель Derived преобразуется в указатель Base), можно использовать явное приведение типов. Поскольку это преобразование безопасное, использование приведения является необязательным условием.

```
Base *pb = (Base*)pd;                   // явное приведение: предупреждает остальных
cout <<" x="<<pb->show()<<endl;         // Метод Base здесь есть
// pb->access(x,y);                     // ошибка: отсутствует в классе Base
```

Это преобразование является безопасным. Компилятор в данном фрагменте программы помечает вызов метода access(x,y) класса Derived как синтаксическую ошибку. Компилятору известно, что указатель pb принадлежит классу Base, а в классе Base отсутствует какой-либо метод access(). Поскольку компилятор C++ не выполняет анализ потока данных, у него есть право не знать то, что известно вам. Компилятор не знает, что указатель pb обозначает вполне развитый объект Derived, который способен ответить на сообщение access(), как и любой другой объект Derived. Разработчик компилятора C++ отдыхает, а нам остается найти метод, чтобы сообщить компилятору то, что известно нам.

Вам известно, как указать контроллеру, что этот небольшой указатель обозначает большой объект. Чтобы сообщить компилятору то, что известно вам, следует воспользоваться приведением. Необходимо выполнить приведение этого слабого указателя Base (который может извлечь возможности класса Derived) к мощному указателю Derived, так что методы Derived станут досягаемыми.

```
Base *pb = (Base*)pd;                   // явное приведение: предупреждает остальных
cout <<" x="<<pb->show()<<endl;         // Метод Base здесь есть
(Derived*)pb->access(x,y) ;             // ошибка: приоритет операторов
```

```
1. Derived pointer, object, and derived method
x = 50 y = 80

2. Derived pointer, derived object, base method
x=50

3. Base pointer, derived object, base method
x=50

4. Converted pointer, derived object and method
x = 50 y = 80

5. Base pointer, base object, base method
x=60

6. Converted pointer, base object, derived method
x=60 y=-33686019
```

Рис. 15.5. *Вывод программы из листинга 15.1*

Это хорошо, однако недостаточно. Операция выбора со стрелкой имеет более высокий приоритет, чем операция приведения. В результате компилятор попытается преобразовать в производный указатель то, что возвращает метод access(). Все перепутывается. Поэтому необходимо использовать еще одну пару скобок.

```
((Derived*)pb)->access(x,y);     // это работает!
```

Это работает, но выглядит устрашающе. Все сложные операции переносятся в программу, поскольку компилятор не может распознать, что указатель pb ссылается на объект Derived.

Объединим все эти компоненты. В листинге 15.1 представлены классы Base и Derived, причем клиентская программа обрабатывает их объекты. Вывод программы показан на рис. 15.5.

Листинг 15.1. Использование указателей для доступа к объектам базового и производного классов

```
#include <iostream>
using namespace std;

class Base {                                    // базовый класс
protected:
    int x;
public:
  Base(int a)                                   // должно использоваться Derived
  { x = a; }
  void set (int a)                              // должно наследоваться
  { x = a; }
  int show () const                             // должно наследоваться
  { return x; } } ;

class Derived : public Base {                   // производный класс
    int y;
public:
  Derived (int a, int b) : Base(a), y(b)
    { }                                         // пустое тело конструктора
  void access (int &a, int &b) const            // добавлено в производный класс
  { a = Base::x; b = y; } } ;

int main()
{ int x, y;
  Derived *pd = new Derived(50,80);             // неименованный производный объект
  cout << " 1. Derived pointer, object, and derived method\n";
  pd->access(x,y);                              // нет проблем: типы соответствуют
  cout <<" x = " <<x <<" y = " <<y <<endl <<end1;              // x = 50 y = 80
  cout << " 2. Derived pointer, derived object, base method\n";
  cout << " x = " << pd->show() << endl << endl;              // x = 50
  Base *pb = pd;                                // указатель на этот же объект
  cout << " 3. Base pointer, derived object, base method\n";
  cout << " x = " << pb->show() << endl << endl;              // x = 50
//  pb->access(x,y) ;                           // ошибка: нет доступа к производному методу
  cout << " 4. Converted pointer, derived object and method\n";
  ((Derived*)pb)->access(x,y) ;                 // знаем, это здесь
  cout << " x = " <<x << " y= " <<y <<endl <<endl;            // x = 50 y = 80
  pb = new Base(60) ;                           // неименованный базовый объект
```

```
cout << " 5. Base pointer, base object, base method\n";
cout << x = " << pb->show() << endl <<endl;                    // x = 60
cout << " 6. Converted pointer, base object, derived method\n";
((Derived*)pb)->access(x,y) ;                          // передается на свой собственный риск
cout <<" x = " <<x <<" y = " <<y <<endl <<endl;                // старье!!
delete pd; delete pb;                                  // необходима аккуратность
return 0 ;
}
```

Вначале клиентская программа создает объект класса Derived и использует указатель класса Derived для доступа к методу access() производного класса. Это тривиально. Компилятор находит метод в определении класса, к которому принадлежит указатель, и вызывает его. При первом выводе печатается x = 50, y = 80.

Затем клиентская программа вызывает метод show() класса Base и использует тот же указатель класса Derived. Это также тривиально. Компилятор не находит определение метода в описании класса Derived, переходит к определению класса Base, находит метод и формирует вызов (и выводит на печать x = 50). Преобразование типов не используется. Неименованный объект, на который ссылался указатель Derived, имеет тип Derived и может выполнить все, что требуется от объекта либо класса Base, либо Derived (второй вывод).

Потом клиентская программа устанавливает указатель Base на объект Derived. Эти указатели разного типа. Обычно неявные преобразования указателей разного типа не допускаются. Поскольку данные указатели являются указателями связанного типа, это правило становится менее строгим для безопасных преобразований.

```
Base *pb = pd;                  // разные типы: безопасно для связанных типов
```

Это безопасно, поскольку указатель класса Derived выполняет все, что может класс Base. Некоторые программисты все еще верят, что явное приведение типов полезно, поскольку оно предоставляет программистам, осуществляющим сопровождение, информацию о преобразовании.

```
Base *pb = (Base*) pd;        // связанные типы: приведение необязательно
```

В листинге 15.1 клиентская программа не использует это преобразование. Затем клиентская программа применяет указатель Base для вызова метода show() класса Base. Поскольку неименованный объект, на который указывает Base, имеет тип Derived, не возникает проблем с отправкой сообщений базового класса к этому объекту. (На печать выводится x = 50, третий вывод.)

Далее клиентская программа использует указатель Base для вызова метода access(). Совершенно не важно, что указатель ссылается на объект класса Derived, который может выполнять задание. Компилятор не проверяет объект, а указатель осуществляет поиск определения класса, к которому принадлежит указатель (Base), не находит согласования с методом и объявляет вызов синтаксической ошибкой. Он превращен в комментарии.

Потом клиентская программа указывает компилятору, что этот базовый указатель ссылается на объект Derived. Клиентская программа выполняет это посредством преобразования указателя Base в указатель Derived. Это преобразование небезопасно и должно выполняться явно. Преобразованный указатель относится к классу Derived, и у компилятора не возникает проблем при вызове метода access().

```
((Derived*)pb)->access(x,y) ;          // известно, это здесь
```

Поскольку объект, на который ссылается данный преобразованный указатель, является объектом класса Derived, вызов метода выполняется правильно и выводит на печать x = 50, y = 80, четвертый вывод.

Клиентская программа создает объект Base и использует указатель Base для вызова метода show() класса Base. Это тривиально. Компилятор осуществляет поиск определения класса, к которому принадлежит указатель, а не класса, к которому относится объект, выполняет согласование сообщения и метода и вызывает его. (На печать выводится x = 60, пятый вывод.)

Наконец, клиентская программа выполняет приведение указателя Base в указатель Derived. Это преобразование не является безопасным, поэтому требуется явное приведение типов, чтобы указать компилятору то, что ему не известно. А именно: программисту известно, что он делает. Приведение указателя выполняется с использованием всех соответствующих скобок и вызывается метод access() класса Derived.

```
((Derived*)pb)->access(x,y);          // передача на свой собственный риск
```

Указание компилятору, что нам известно о происходящем, означает, что методы класса Derived не будут вызываться с помощью этого указателя, поскольку объект, на который он ссылается, может выполнять только работу класса Base.

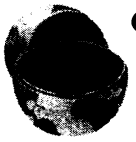
В этом случае компилятор не пытается предугадывать действия программиста и вычислять, на какой вид объекта реально указывает Base. Об этом можно сожалеть, поскольку метод access() класса Derived, вызываемый в объекте Base, выводит непонятно что (шестой вывод).

Рассмотрим три комментария. Все, что было здесь сказано об указателях, также справедливо и в отношении ссылок C++. (Единственное отличие состоит в том, что ссылку невозможно изменить так, чтобы она указывала на другой объект.) Ссылка базового типа может указывать на объект производного класса без выполнения приведения и вызывает только методы, определенные в базовом классе, но не в производном классе. Ссылка производного класса не может указывать на объект базового класса без выполнения явного приведения типов. С использованием приведения она может указывать на базовый класс и вызывать любой метод. Программист должен показать, что метод производного класса не вызывается с использованием ссылки производного класса, которая указывает на объект базового класса.



Внимание Указатели (и ссылки) базового класса могут указывать на объект производного класса без явного приведения. Они не вызовут каких-либо повреждений, потому что могут осуществлять доступ только к базовой части производного объекта. Явное приведение типов является необязательным. Указатели (и ссылки) производного класса не должны указывать на объект базового класса, потому что они должны запрашивать от объекта отклик на сообщения производного класса. При необходимости используйте явное приведение.

Все сказанное здесь об указателях и ссылках выполняется только для связанных классов. Если классы не связаны наследованием, их указатели (и ссылки) не могут указывать на объекты других классов без явного наследования. Неявные преобразования не допускаются, потому что классы не имеют общих операций. Единственная причина, по которой допускается, чтобы базовый указатель ссылался на производный объект, заключается в том, что базовый и производный классы имеют общие операции — операции, определенные в базовом классе. Базовый указатель может вызвать эти операции.



Осторожно! Указатели (и ссылки) конкретного класса не должны указывать на объекты классов, не связанных с ним наследованием. Это синтаксическая ошибка. При необходимости используйте явное приведение. Язык C++ это допускает.

Неявные приведения классов, связанных наследованием, допускаются в случае, если режим наследования является общедоступным. Если наследование закрытое или защищенное, то все преобразования требуют явного приведения типов. Подобные действия связаны с тем, что в закрытом или защищенном режиме наследования общедоступные операции базового класса становятся в производном классе закрытыми или защищенными. Отсутствуют гарантии, что базовые и производные классы имеют какие-либо общие операции. Следовательно, базовому указателю должно быть запрещено указывать на объект производного класса. Операции производного класса не доступны базовому указателю (главное свойство классов C++), а операции базового класса — производному объекту (свойства закрытого и защищенного наследования). Рекомендуем использовать только открытый режим порождения (derivation).



Советуем Осуществляйте порождение производных классов, используя только открытый вывод. Это позволит нацеливать указатели (или ссылки) одного класса на объект другого класса в одной иерархии наследования. В закрытом и защищенном наследовании базовые и производные объекты не содержат общедоступных операций (или данных).

Преобразование аргументов указателя и ссылки

Поговорим о преобразовании аргументов при вызове функций. У нас есть класс Other, который реализует функции-члены с параметрами в виде указателя и ссылки из классов Base и Derived.

Перегруженные методы setB() ожидают параметры класса Base. Они устанавливают значение Other элемента набора данных в значение элемента набора данных в параметре Base. Перегруженные методы setD() ожидают параметры класса Derived. Они устанавливают элемент данных Other в значение дополнительного элемента данных в параметре Derived объекта. Метод get() возвращает внутреннее состояние объекта Other.

```
class Other {                          // другой класс
  int z ;
public:
  void setB(const Base &b)             // передача по ссылке
  { z = b.show(); }
  void setB(const Base *b)             // передача по указателю
  { z = b->show(); }
  void setD(const Derived &d)          // передача по ссылке
  { int a; d.access(a,z); }
  void setD(const Derived *d)          // передача по указателю
  { int a; d->access(a,z); }
  int get() const                      // селектор
  { return z; } } ;
```

Этого достаточно, чтобы продемонстрировать главные вопросы. В следующем фрагменте программы каждая функция получает аргумент типа, определенного интерфейсом функции. Такой способ для использования функции является наиболее естественным. Функция ожидает аргумент определенного типа. Имейте в виду, что внутри параметра функции аргумент принимает сообщения, которые принадлежат соответствующему типу. В функциях setB() параметр отправляет сообщение show() класса Base, в функциях setD() — сообщение access() класса Derived.

```
Base b(30); Derived d(50,80);          // связанные объекты
  Other a1, a2;                        // несвязанные объекты
  a1.setB(b); a2.setD(d);              // точное согласование
  a1.setB(&b); a2.setD(&d) ;           // точное согласование
```

В дополнение к сообщениям, определенным в классе Derived, функции, ожидающие в качестве параметра ссылку класса Derived, могут отправить в параметре сообщения, которые определены также в классе Base. Это не проблема, поскольку аргумент класса Derived может отвечать на сообщения, наследованные из класса Base (наследование в данном случае является общедоступным, а не защищенным или закрытым).

В функциях, которые ожидают параметр класса Base (ссылку или указатель), сообщения класса Derived не могут быть отправлены параметру. Откуда это известно? Поскольку параметр класса Base не может ответить на сообщения, определенные в классе Derived, попытка отправить такое сообщение внутри функции компилироваться не будет.

```
class Other {                          // другой класс
  int z ;
public:
  void setB(const Base &b)             // ожидается объект Base
  { int a; d.access(a,z); }            // ошибка: сообщение Derived
  . . . } ;                            // остальная часть класса Other
```

Эти два результата позволяют решить, что случится, если указатель или ссылка другого класса передается как реальный аргумент функции. Если параметр и тип аргумента не связаны наследованием, ответ простой. Если отсутствует явное приведение типов, то вызов функции является синтаксической ошибкой, независимо от того, передается параметр ссылкой или указателем.

```
Account acc1(100), acc2(1000);         // несвязанные объекты
Other a1, a2;                          // несвязанные объекты
a1.setB(acc1); a2.setD(acc2);          // синтаксическая ошибка
a1.setB(&acc1); a2.setD(&acc2);        // синтаксическая ошибка
```

Параметры передаются значением. Если класс параметров предусматривает соответствующие конструкторы, возможно неявное приведение. Для параметров ссылок и указателей такой механизм не существует. Единственным способом, который позволяет исключить синтаксические ошибки, является использование явного приведения.

```
a1.setB((Base&)acc1) ;          // не синтаксическая ошибка, но бессмысленно
a1.setB((Base*)&acc1);          // не синтаксическая ошибка, но бессмысленно
```

Эти вызовы функций не содержат синтаксических ошибок — компилятор думает, что вы уверены в своих действиях. Вам же неизвестно, что происходит. В данных функциях параметр Account объектов собирается отвечать на сообщения Base (в данном случае show()). Это является семантической ошибкой: действия программы не имеют смысла.

Когда формальный параметр и типы фактических аргументов связаны друг с другом наследованием, ситуация еще больше осложняется.

Функцию ожидающую указатель Base (или ссылку Base), можно вызвать с указателем (или ссылкой) Derived как фактическим аргументом. Это замечательно, потому что объект Derived может делать все то же самое, что и объект Base. Функция, ожидающая параметр Base, запрашивает от своих параметров выполнения в теле функции только обязанностей Base. Как уже говорилось, сообщения класса Derived внутри тела этой функции будут недопустимыми для компилятора.

```
void Other::setB(const Base &b)        // передача по ссылке
{ int a; b.access(a,z); }              // синтаксическая ошибка
```

Эта ситуация невозможна, потому что C++ является языком со строгим контролем за типами. Следовательно, преобразование из указателя (ссылки) Derived

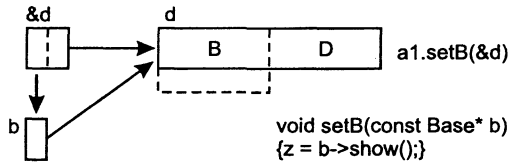


Рис. 15.6. *Преобразование указателя из Derived в Base при передаче параметра*

в указатель (ссылку) Base безопасно. С объектом аргумента Derived внутри функции, ожидающей объект Base, ничего плохого не случится.

```
a1.setB(&d) ;        // безопасное преобразование
```

На рис. 15.6 показан вызов данной функции. Когда в памяти выделяется область для параметра указателя b, он инициализируется в содержание фактического аргумента (толстая стрелка). Фактический аргумент является неименованным указателем на объект d класса Derived. Этот безымянный указатель обозначен как &d. В результате оба указателя ссылаются на тот же объект класса Derived (тонкие стрелки). Когда функция выполняется, она отправляет сообщения своему параметру b. Поскольку это параметр класса Base, он может извлечь сообщения только из части Base объекта (пунктирная линия под объектом). Указатель параметра не может получить сообщения из части Derived объекта, но эти сообщения не вызываются внутри функции, потому что его параметр принадлежит классу Base, а не классу Derived.

Когда у функции имеется параметр-указатель (или ссылка) Derived, не следует вызывать эту функцию с указателем (или ссылкой) Base как фактическим аргументом. В теле такой функции от объекта, указанного параметром указателем, можно потребовать выполнения того, что может сделать только мощный объект Derived, но не слабый объект Base.

```
a2.setD(&b) ;        // синтаксическая ошибка
```

Это преобразование не является безопасным и помечается как синтаксическая ошибка. На рис. 15.7 показан вызов функции. Когда выделяется пространство для параметра d, он инициализируется содержанием реального аргумента, неименованного указателя на объект b класса Base. Когда выполняется функция, она отправляет сообщения своему параметру d. Параметр относится к классу Derived, поэтому он может извлекать сообщения как из Base, так и из Derived частей объекта (пунктирная линия).

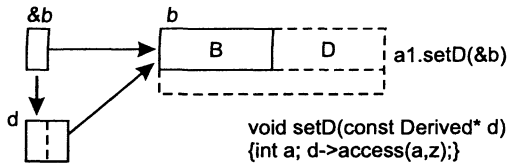


Рис. 15.7. *Преобразование из указателя Base в указатель Derived при передаче параметра*

Но этот небольшой объект не содержит какой-либо части Derived. Слабый объект Base не знает, как на них отвечать. Когда во время выполнения несуществующей части объекта направляется сообщение, результат не определяется. Программа может завершиться аварийно или выдать неверные результаты.

Предположим, что метод Derived::setD() написан иначе и отправляет своему параметру только сообщения класса Base.

```
void setD(const Derived *d)          // передача по указателю
{ z = d->show(); }                   // только сервисы Base
```

Фактически отправление объекта Base этой функции является безопасным, но компилятор этого не знает. Можно применить явное приведение, чтобы сообщить компилятору то, что нам известно.

```
a2.setD((Derived*)&b) ;              // явное преобразование
```

Это обычный способ сообщения компилятору, что вам известно о своих действиях. Когда данное приведение встречается впервые, оно кажется сложным. Однако оно просто указывает, что этот объект Base будет безопасен в рамках функции setD(). Итак, этот небольшой объект Base может претендовать на то, что он является выросшим объектом Derived, потому что внутри этой функции он в любом случае будет выполнять только работу Base. Но обязательно следует убедиться в правильности ваших действий!

В листинге 15.2 обобщаются результаты этого обсуждения. Вывод программы представлен на рис. 15.8.

Листинг 15.2. Передача параметров указателя и ссылки базового и производного классов

```
#include <iostream>
using namespace std;

class Base                                          // базовый класс
protected:
    int x;
public:
  Base(int a)                                       // используется Derived
  { x = a; }
  void set (int a)                                  // наследуется
  ( x = a; }
  int show () const                                 // наследуется
  { return x; } } ;

class Derived : public Base {                       // производный класс
    int y;
public:
  Derived (int a, int b) : Base(a), y(b)
    { }                                             // пустое тело конструктора
  Derived(const Base &b) : Base(b)                  // поддерживает явное приведение типов
  { y = 0; }                                        // явная инициализация
  void access (int &a, int &b) const                // добавлено в производный класс
  { a = Base::x; b = y; } } ;

class Other (                                       // другой класс
  int z ;
public:
  void setB(const Base &b)                          // передача по ссылке
  { z = b.show(); }
  void setB(const Base *b)                          // передача по указателю
  { z = b->show(); }
  void setD(const Derived &d)                       // передача по ссылке
  { int a; d.access(a,z); }
  void setD(const Derived *d)                       // передача по указателю
  { int a; d->access(a,z); }
  int get() const                                   // средство доступа
  { return z; }
} ;

int main()
{ Base b(30); Derived d(50,80);                     // связанные объекты
  Other a1, a2;                                     // несвязанные объекты
  a1.setB(b); a2.setD(d);                           // точное согласование
  cout << " a1=" << a1.get() << " a2=" << a2.get() << endl;
  a1.setB(d); a2.setD(b);                           // неявное преобразование
  cout << " a1=" << a1.get() << " a2=" << a2.get() << endl;
  a1.setB(&b); a2.setD(&d);                         // точное согласование
  cout << " a1=" << a1.get() << " a2=" << a2.get() << endl;
  a1.setB(&d);                                      // неявное преобразование
//  a2.setD(&b);                                    // синтаксическая ошибка
  a2.setD((Derived*)&b) ;                           // явное преобразование
  cout << " a1=" << a1.get() << " a2=" << a2.get() << endl;
  return 0 ;
  }
```

В этом примере используются те же классы Base и Derived, что и в предыдущих примерах. В классе Derived имеется дополнительный конструктор.

```
a1=30 a2=80
a1=50 a2=0
a1=30 a2=80
a1=50 a2=7011896
```

Рис. 15.8. *Вывод программы для листинга 15.2*

Класс Other располагает двумя перегруженными функциями setB(), которые ожидают параметр-ссылку и указатель класса Base, и двумя перегруженными функциями setD(), которые ожидают параметр-ссылку и указатель класса Derived, а также методом get(), возвращающим значение элемента набора данных z.

Клиентская программа определяет и инициализирует объект Base, объект Derived и два объекта Other. Первая строка вывода дает a1 = 30 и a2 = 80, потому что вызовы функции setB() и setD() используют точное согласование типов фактических аргументов и формальных параметров.

При вызове функции setB() с фактическим аргументом класса Derived не возникает проблем. Ссылка Base может быть инициализирована объектом Derived без выполнения приведения типов, поскольку это преобразование безопасно. Обычно подобный вызов функции должен быть отклонен как синтаксическая ошибка. Вызов можно сделать допустимым для компилятора при использовании явного приведения.

```
a2.setD((Derived&)b);          // синтаксис приведения типов ссылок
```

Данное приведение ничего хорошего не дает, потому что оно просто успокаивает компилятор. Ссылка Derived в setD() все еще указывает на небольшой объект Base (см. рис. 15.7). Вызов функцией Derived::access() в теле setD() осуществляет доступ к памяти, которая не относится к параметру объекта b.

Чтобы этот вызов имел смысл, параметр указателя Derived в setD() должен ссылаться на объект Derived, а не на Base. Объект Derived должен инициализироваться значениями, которые содержатся в полях фактического аргумента Base. Однако в объекте Derived имеются поля, отсутствующие в объекте Base. Они должны быть установлены в некоторые приемлемые значения, например в 0, или в любое другое, отвечающее требованиям приложения.

Что должна обеспечить функция для класса Derived, чтобы гарантировать надлежащую инициализацию полей объекта? Надо использовать конструктор. Его имя зависит от количества и типа параметров. Поскольку этот конструктор инициализирует поля объекта Derived, используя данные объекта Base, ему требуется только один параметр — типа Base. Следовательно, это должен быть конструктор преобразования.

Как можно видеть, имя хорошо подходит конструктору, который преобразует значение класса Base в значение класса Derived.

```
Derived::Derived(const Base &b) : Base(b)        // копирование конструктора
{ y = 0; }                                       // явная инициализация
```

Конструктор преобразования может вызываться явно и создавать временную переменную класса Derived, которая инициализируется конструктором, указывается параметром ссылкой внутри setD() и удаляется после вызова setD().

```
a2.setD((Derived)b);           // явный вызов конструктора
```

Подобную методику можно использовать только для входных параметров. Если она применяется для параметра вывода, то изменения внутри функции будут сделаны во временной копии, уничтожаемой после завершения вызова.

Поскольку этот параметр не определен как явный, его можно вызывать неявно, что осуществляется в клиентском вызове в листинге 15.2. Следующая строка вывода дает a1 = 30 и a2 = 0.

Вторая часть клиентской программы работает с параметрами-указателями, а не с параметрами-ссылками. Первые два вызова setB() и setD() используют

точное согласование и дают вывод a1 = 30 и a2 = 80. Вызов setB() с указателем Derived в качестве фактического аргумента не создает никаких проблем. Преобразование из Derived в Base является безопасным в отличие от вызова setD() с указателем Base в качестве аргумента. Это преобразование небезопасное и помечается как синтаксическая ошибка.

Чтобы убедить компилятор принять вызов, используется явное приведение типа указателя из Base в Derived. Компилятор успокаивается, но объект Derived он не создает. Решить такую проблему мог бы конструктор с параметром-указателем (подобным использованному для параметра ссылки). Но эти конструкторы не являются общепринятыми, и поэтому в примере использован старый фрагмент, чтобы еще раз показать небезопасность приведения из Base в Derived.



Внимание Всегда безопасно передавать указатель (или ссылку) производного класса как аргумент для функции с параметром-указателем (или ссылкой) базового класса (приведение является необязательным). Передача базового указателя (или ссылки) как аргумента для функции с параметром-указателем (или ссылкой) производного класса является синтаксической ошибкой. Принуждение компилятора принять вызов этой функции, используя приведение базового указателя (или ссылки) к производному классу, может быть опасным процессом.

Мы подробно рассмотрели вопросы преобразования объектов (указателей либо ссылок) разных классов. Для несвязанных классов эта тема не имеет какого-либо практического значения. Редко случается, что вы можете использовать некоторый тип, когда ожидается другой несвязанный тип. Преобразования несвязанных типов сбивают с толку и опасны.

Совершенно другое дело для классов, связанных наследованием. Использование одного типа вместо ожидаемого другого типа из общей иерархии наследования популярно при программировании на C++ (как, впрочем, и в любом объектно-ориентированном программировании). Важно понимать этот метод, его ограничения и последствия.

Преобразования типов, связанных наследованием, также вызывают путаницу и могут быть опасными. Они не подчиняются обычной интуиции программирования в отношении преобразований числовых типов. Главным критерием такого преобразования является не наличие достаточного пространства у результата преобразования для размещения источника, а как раз наоборот. Главное — является ли преобразование безопасным и будет ли от его результата зависеть выполнение чего-то такого, что оно не может сделать.

Преобразование из производного типа в базовый является безопасным. Преобразование из базового типа в производный — опасно.

Рассмотрим виртуальные функции C++.

Виртуальные функции

В C++ каждый объект вычислений характеризуется свойствами, которые определяют тип объекта. Объект обозначается его именем (идентификатором) и типом, связанным с этим идентификатором. Вы встречались с подобной связью, когда определяли типы объекта в объявлениях. Это справедливо как для переменных, так и для функций программ.

При объявлении или определении переменной исходная программа должна выполнить связывание и определить тип объекта вычислений. Эта связь между именем объекта и его типом не может быть разорвана во время выполнения. Программа может определять другие объекты вычислений, используя этот же идентификатор и тип. Но другие объекты будут другими объектами вычислений, даже если они используют одно и то же имя.

Это же справедливо и для функций программ. Объявление функции (ее прототипа) или определение функции (ее тела) включает идентификатор имени функции. Имя функции связывается с объектной программой, сгенерированной для этой функции. Подобная связь не может быть разорвана или изменена во время выполнения программы. Программа может определить некоторые другие функции, используя это же имя функции. Функции могут быть в одном и том же классе (с разными сигнатурами) или в другом классе или области видимости (с той же или с иной сигнатурой). Но это будут другие функции. Просто у них одинаковое имя функции.

Перегрузка имени функции C++ не разрывает связь языка C с уникальными именами функции. Для человека, когда имя функции повторно используется в том же классе или в другой области видимости, это то же самое имя. Для компилятора все эти функции будут иметь разные имена. Имя, известное компилятору, является конкатенацией имени класса, к которому принадлежит функция, идентификатора функции, типа результата и типов параметров.

Когда компилятор видит определение функции, он создает видоизмененные имена. В них к идентификатору функции добавляются имя класса, тип результата и типы параметров. В результате для компилятора каждое имя функции действительно уникально. Этот метод называется искажением имен.

Например, функция `draw()` может быть определена в нескольких классах и с различными сигнатурами. Каждая функция представляет собой отдельный объект вычисления.

```
class Circle {
  int radius;
public:
  void draw() ;
  . . . . } ;                    // остальная часть класса Circle

class Square {
  int side;
public:
  void draw();
  ....};                         // остальная часть класса Square

class Rectangle {
  int sidel, side2;
public:
  void draw();
  . . . . } ;                    // остальная часть класса Rectangle
```

Связь между именем и объектом вычислений устанавливается во время компиляции, когда компилятор обрабатывает определение или объявление объекта вычислений. Рассмотрим, например, следующий фрагмент клиентской программы, определяющий объекты `Circle`, `Square` и `Rectangle` и рисующий их на экране.

```
Circle c; Square s; Rectangle r;          // имя/тип связываются
c.draw(); s.draw(); r.draw();             // имя/функция объединены в пару
```

Компилятор и лицо, осуществляющее сопровождение, знают, что объект `c` имеет тип `Circle`, объект `s` — тип `Square`, а объект `r` — тип `Rectangle`. Компилятор и сопровождающее лицо уверены в том, что первый вызов `draw()` указывает на `Circle::draw()`, второй вызов `draw()` относится к `Square::draw()`, а третий вызов `draw()` — к `Rectangle::draw()`.

В языках, подобных C++, нежелательно изменять эти связи на этапе компиляции во время выполнения программы. Тип объекта в программировании описывает фиксированные свойства объекта. Это еще одно проявление строгого контроля типов.

В настоящее время строгий контроль типов в выражениях и при передаче параметров рассматривается как доказанный тип. Для каждого объекта вычислений набор допустимых операций для объекта известен заранее как компилятору, так и проектировщику клиентской программы и лицу, осуществляющему сопровождение.

Строгий контроль типов обеспечивает так называемое раннее связывание. Тип объектов вычислений фиксируется на этапе компиляции и не изменяется во время выполнения программы. Другой популярный термин — статическое связывание. Оно означает то же самое. Связь между именем объекта и типом объекта фиксируется при компиляции и не может изменяться динамически во время выполнения программы.

Когда сообщение, определенное его именем и списком фактических аргументов, посылается объекту, компилятор интерпретирует его в соответствии с классом (типом) этого объекта. Имя класса объекта известно на этапе компиляции и не изменяется во время выполнения.

Статическое связывание является стандартом таких современных языков, как C++, C, Java, Ada, Pascal (но не Lisp). Вначале оно было введено для повышения производительности, а не для улучшения качества программы. Динамическое связывание, поиск значения вызова функции во время выполнения отнимает время. Когда значение вызова функции фиксируется при компиляции, программа выполняется быстрее.

Позднее обнаружилось, что статическое связывание может с успехом использоваться для осуществления контроля типов. Если функция вызывается с неверным количеством или неправильными типами аргументов, этот вызов отклоняется на этапе компиляции. Если имя сообщения (с соответствующей сигнатурой) не обнаружено в спецификациях класса, вызов отклоняется при компиляции, а не во время выполнения.

Строгий контроль типов предусматривает контроль типов на этапе компиляции и повышает производительность во время выполнения. Это полезно в большинстве приложений.

Когда еще вам хотелось бы установить связь между идентификатором и объектом вычислений? Можно ответить: на этапе выполнения.

Рассмотрим обработку неоднородного списка объектов или обработку внешнего входного потока с объектами разных типов. При файловом вводе или вводе от интерактивного пользователя программа не знает точно тип объекта, поступающего из внешней среды.

Например, программа может выводить на экран изображения, показывая составляющие их фигуры одну за другой. Программа должна вызвать `Circle::draw()`, `Square::draw()`, `Rectangle::draw()` или еще какую-либо известную ей фигуру. Это было бы замечательно. Однако лучше всего использовать только один оператор в исходной программе и изменять его значение в зависимости от фактической природы объекта `shape`.

```
shape.draw();          // из класса Circle, Square или Rectangle
```

Если формой объекта в текущем проходе цикла является `Circle`, то этот оператор вызывает `Circle::draw()`. Если — `Square`, то следует задать `Square::draw()`. Если это `Rectangle`, вызовите `Rectangle::draw()`.

При строгом контроле за типами это невозможно. Компилятор найдет объявление переменной `shape`, установит ее класс и проверит определение класса. Если в этом классе не будет найдена пустая функция `draw()` с отсутствующими параметрами, компилятор генерирует сообщение об ошибке. Если функция обнаружена, компилятор генерирует объектный код. Но тип функции `draw()` во время компиляции будет фиксированным. Во время выполнения уже будет невозможно осуществлять поиск значения функции `draw()`.

То, что нам в данный момент нужно, называется связыванием этапа выполнения или поздним динамическим связыванием. Предположим, что существует несколько объектов вычисления (функций `draw()` в различных классах). Требуется связать один из этих объектов вычислений с именем в вызове конкретной функции. Более того, желательно, чтобы эта функция `draw()` в представленном вызове функции означала `Circle::draw()`, `Square::draw()`, `Rectangle::draw()` и т. д. Хотелось бы, чтобы это значение устанавливалось не на этапе компиляции, а при выполнении. Тогда различные фигуры будут нарисованы в зависимости от значения вызова функции.

Поговорим о терминологии. Техническим термином для установки значения имени функции является связывание (binding). Компилятор связывает имя функции с конкретной функцией. Желательно, чтобы это связывание имело место на этапе выполнения. Поэтому оно называется динамическим связыванием, а не связыванием при компиляции. Нужно, чтобы связывание происходило после этапа компиляции, поэтому оно называется поздним, а не ранним, связыванием. Требуется, чтобы такое связывание допускало присваивание различных значений тому же самому имени функции в зависимости от природы используемого объекта. Именно поэтому оно называется динамическим, а не статическим связыванием.

Способность имени функции принимать при своем вызове различные значения называется полиморфизмом (от "множество форм"). Некоторые авторы используют термин "полиморфизм" в более широком смысле, включая использование одного имени функции в различных классах, но без динамического связывания. Обратите внимание, что при использовании полиморфизма подразумевается позднее или динамическое связывание, присвоение значения вызова метода на этапе выполнения в зависимости от фактического типа объекта, т. е. назначение сообщения.

Все это должны обеспечить виртуальные функции C++.

Динамическое связывание: традиционный подход

Динамическое связывание не является каким-то особенным вопросом для объектно-ориентированного программирования. Обработка неоднородных списков всегда была обычной вычислительной задачей, и программисты привыкли реализовывать динамическое связывание на любом языке. Наша задача заключалась в обработке подобных объектов. Они настолько подобны, что имеет смысл использовать одно имя для функции во всех категориях объектов (например, `draw()`). Но типы объектов не являются идентичными — каждая функция выполняет заданные действия по-своему.

Как пример рассмотрим обработку списка записей в базе данных университета. Предположим, что существует только два типа записей: для студентов и преподавателей. Допустим, что программа сохраняет лишь три части информации: идентификатор университета, наименование и либо служебное положение (для преподавателей), либо специализацию (для студентов). Краткий пример данных представлен на рис. 15.9.

```
FACULTY
U12345678
Smith, John
Associate Professor
1
STUDENT
U12345611
Jones, Jan
Computer Science
FACULTY
U12345689
Black, Jeanne
Assistant Professor
STUDENT
U12345622
Green, James
Astronomy
```

Рис. 15.9.
Входные данные для примера динамического связывания

Длина значения идентификатора одинакова для каждого лица (девять символов) и может быть реализована как массив символов фиксированной длины. Имя, служебное положение и специализация имеют разные длины для разных лиц. Следовало бы реализовать их как динамически выделенные массивы. Структура для отдельного лица выглядит следующим образом.

```
struct Person {
       int kind;            // 1 для преподавателей, 2 для студента
       char id[10];         // фиксированной длины
       char* name;          // переменной длины
       char* rank;          // только для преподавателей
       char* major; } ;     // только для студента
```

Эту структуру можно реализовать как класс с конструкторами, деструктором и функциями-членами. Но на данном этапе нам непонятно, о чем идет речь. Эти элементы будут введены при обсуждении более современного подхода.

В первом традиционном подходе характеристики различных видов объектов объединяются (например, служебное положение, специализация) в один класс. Чтобы обработать каждый вид объектов по-разному, добавляется поле для описания, к какой области принадлежит конкретный объект. В клиентской программе используются либо операторы выбора `switch`, либо операторы `if`, ветви которых реализуют обработку различных видов объектов.

Вместо определения полей для обоих видов объектов можно было бы использовать конструктор `union`. Для массивов фиксированного размера это имеет смысл. Но при динамическом управлении памятью из-за этого могут возникнуть дополнительные сложности.

Динамическое управление памятью сохраняет пространство и предотвращает переполнение памяти. Простым методом сохранения данных в памяти является определение массива объектов `Person`. Хотя и используется зарезервированное слово `struct`, переменные типа `Person` являются объектами, потому что в C++ зарезервированные слова `struct` и `class` — синонимы (за исключением прав доступа по умолчанию и наследования по умолчанию).

```
Person data [1000];        // массив входных данных
```

Рекомендуем вам хранить данные в массиве указателей на объекты, а не в массиве объектов. Для выделения большого массива указателей не требуются большие затраты памяти. В случае переполнения массив указателей можно повторно выделить, не копируя существующие данные (см. главу 6). Пространство для каждого объекта `Person` будет выделено после считывания данных этого объекта из входного файла.

```
Person* data [1000];       // массив указателей
```

Для считывания данных из входного файла определен объект `ifstream` библиотечного класса. Он всегда открывается для ввода. Для ассоциации физического файла с объектом логического файла имя физического файла должно быть определено как параметр вызова конструктора.

```
ifstream from( "univ.dat");                    // файл входных данных
if (!from) { cout << " Cannot open file\n"; return 0; }
```

Для каждого объекта входного файла программа динамически выделяет структуру, а затем считывает четыре элемента данных: строку, определяющую тип объекта, идентификатор, имя и либо служебное положение (для преподавателей), либо специализацию (для студентов). Программа проверяет значение строки, определяющей тип объекта ("FACULTY" или "STUDENT"), и устанавливает поле типа объекта либо в 1, либо в 2.

```
char buf[80];                      // буфер входных данных
Person *p = new Person;            // выделение памяти для нового объекта
from.getline(buf,80) ;             // распознавание поступающего типа
if (strcmp(buf, "FACULTY") == 0)
  p->kind = 1;                     // 1 для преподавателей
```

```
else if (strcmp(buf, "STUDENT") == 0)
  p->kind = 2;                          // 2 для студента
else
  p->kind = 0;                          // тип не известен
```

Поскольку длина поля идентификатора известна, его можно непосредственно считать в поле объекта Person. Длина данных для имени, служебного положения и специализации неизвестна до тех пор, пока данные не будут считаны в память. Следовательно, программа должна считать данные в буфер фиксированного размера, измерить длину данных, выделить достаточную память в динамически распределяемой области памяти и скопировать данные из буфера в память динамически распределяемой области.

```
from.getline(p->id,10);                 // считывание id
from.getline(buf,80);                   // считывание имени
p->name = new char[strlen(buf)+1];      // выделение памяти
strcpy(p->name, buf);                   // копирование имени
from.getline(buf,80);           // чтение служебное положение/специализация
if (p->kind == 1)
  { p->rank = new char[strlen(buf)+1];  // память для служебного положения
    strcpy(p->rank, buf); }             // копирование служебного положения
else if (p->kind == 2)
  { p->major = new char[strlen(buf)+1] ;      // память для специализации
    strcpy(p->major, buf); }            // копирование специализации
```

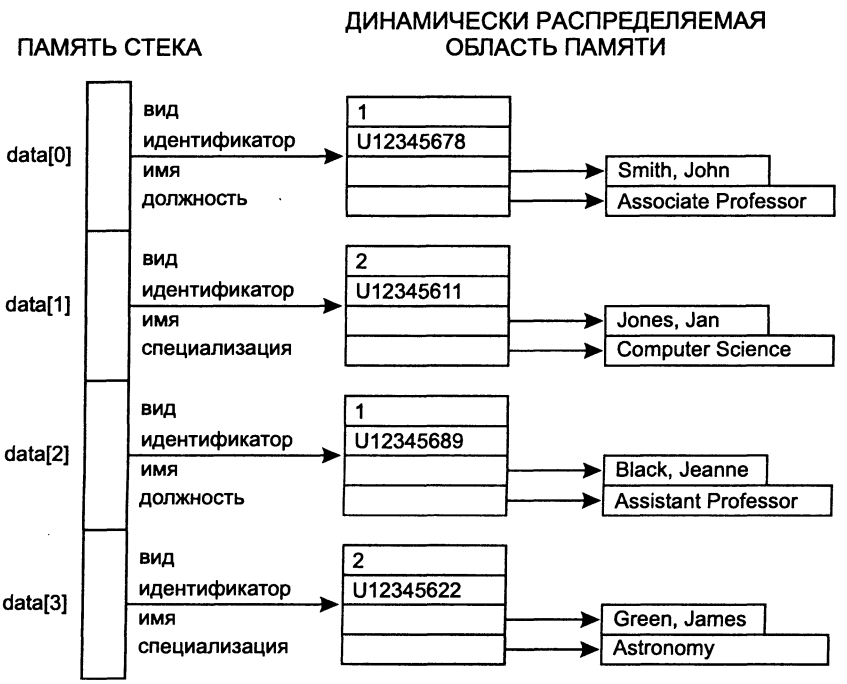


Рис. 15.10. *Структура динамически распределенной памяти для входных данных*

На рис. 15.10 представлена структура элемента данных для этого примера. Массив data[] в левой части рисунка является стековым массивом, а вся остальная память справа от массива (объекты типа Person и их динамическая память) выделяются из динамически распределяемой области памяти.

Хорошей идеей считается инкапсуляция алгоритма считывания в функцию, например read(), чтобы клиентская программа передавала этой функции файловый

объект и указатель Person. Функция read() должна выделять память для объекта Person, считывать данные из файла и заполнять объект Person входными данными.

```
Person* data[20]; int cnt = 0;           // массив указателей
ifstream from("univ.dat");               // входной файл: библиотечный объект
if (!from) { cout << " Cannot open file\n"; return 0; }
while (!from.eof())                      // считывание до eof
{ read(from, data[cnt]);                 // data[cnt] имеют тип
Person*
  cnt++; }
cout << " Total records read: "<< cnt << endl << endl;
```

Соберем функцию read() из частей, описанных ранее. У этой функции есть два главных недостатка, которые относятся к передаче параметра.

```
void read (ifstream f, Person* person)        // плохой интерфейс
{ char buf[80];
  Person* p = new Person;                 // выделение памяти для нового объекта
  f.getline(buf,80);                      // распознавание входного типа
  if (strcmp(buf, "FACULTY") == 0)
    p->kind = 1;                          // 1 для преподавателей
  else if (strcmp(buf, "STUDENT") == 0)
    p->kind = 2 ;                         // 2 для студента
  else
    p->kind = 0;                          // тип неизвестен
  f.getline(p->id,10);                    // считывание идентификатора
  f.getline(buf,80);                      // считывание имени
  P->name = new char[strlen(buf)+1];      // выделение памяти
  strcpy(p->name, buf);                   // копирование имени
  f.getline(buf,80) ;       // считывание служебного положения/специализации
  if (p->kind == 1)
    {  p->rank = new char[strlen(buf)+1]; // память для служебного положения
       strcpy(p->rank, buf); }          // копирование служебного положения
  else if (p->kind == 2)
    {  p->major = new char[strlen(buf)+1];    // память для специализации
       strcpy(p->major, buf); }               // копирование специализации
  person = p; }                               // присоединение к массиву
```

Параметры передаются функции по значению. Это очевидно в случае файлового объекта. Когда данные считываются из файла, внутреннее состояние файлового объекта изменяется. Если внутреннее состояние остается без изменений, то в следующий раз файл считает эти же данные, а не следующую запись. Когда файловый объект передается по значению, внутреннее состояние объекта параметра изменяется, а внутреннее состояние параметра файлового объекта останется тем же. Объекты не следует передавать по значению. Они должны передаваться по ссылке.

```
void read (ifstream& f, Person* person)      // считывание одной записи
{ char buf[80] ;
  Person* p = new Person;                 // выделение памяти для нового объекта
  . . .                                   // остальная часть read()
  person=p; }                             // присоединение нового объекта
```

Рекомендуем помечать ссылки на объект как константы, если функция не изменяет состояние объекта во время ее выполнения. В данном примере модификатор const отсутствует, потому что при считывании информации из файла файловый объект изменяется. Программист серверного класса ifstream клиентской части не должен проектировать серверный класс. Чтобы передать операции ввода/вывода, ему достаточно знать, что состояние файлового объекта изменяется.

Теперь опишем параметр-указатель. Если в C++ параметр передается указателем (или по ссылке), то его значение может изменяться в пределах функции и оно будет оказывать влияние на фактический аргумент в клиентском пространстве. Утверждение о том, что "параметр передается по указателю", должно восприниматься серьезно. Оно не обозначает "параметр-указатель" (именно это чаще всего запутывает программистов). В варианте функции read() указатель на некоторое лицо Person передается по значению. Следовательно, его значение не может изменяться при вызове функции. Если до вызова функции параметр-указатель был направлен в никуда, он будет указывать в неопределенном направлении и после вызова функции, а не на выделенный объект Person.

Здесь с помощью указателя передается объект Person. Действительно, если объект Person надлежащим образом передается в эту функцию, он будет правильно заполнен данными входного файла.

```
Person person;                  // объект Person, не указатель
read(from, &person);            // объект передается по указателю
```

"Надлежащая" передача означает передачу адреса объекта. Такой объект существует в клиентском пространстве. Передача объекта по указателю позволяет программе изменять ее при вызове функции. Даже в этом случае функция read() сталкивается с проблемой: переменная person теперь имеет тип Person, а у переменной p, используемой в последней строке read(), есть тип Person*. Эти два типа очень похожи, однако они немного различаются. Они относятся к разным типам. Один является классом со всеми своими членами, другой — указателем на объект класса. Это можно зафиксировать, если необходимо. Однако это оставило бы нас в клиентском пространстве с массивом объектов Person вместо массива указателей Person.

Вам надо зафиксировать функцию read() для уверенности, что вся память динамически распределяемой области, выделенная в read(), правильно присоединена к памяти стека, показанной на рис. 15.10.

В момент обращения функции к read() объекта Person еще не было. Теперь он выделяется в функции. Это указатель на объект Person, который существует в клиентском пространстве (весь массив указателей в целом). Указатель, который передается функции read().

```
while (!from.eof())                 // считывание до eof
{ read(from, data[cnt]);            // data[cnt] типа Person*
  cnt++; }
```

Перед вызовом фактический аргумент, указатель Person, содержит "мусор", поскольку он является частью массива динамически распределяемой области памяти. После вызова указатель ссылается на объект Person, выделенный в функции read(), как показано на рис. 15.10. Следовательно, он содержит действительный адрес участка верхней части памяти и его содержимое изменяется после вызова. Затем указатель должен передаваться либо по ссылке, либо по указателю. Именно поэтому вариант read(), представленный выше, неверен.

Убедитесь, что вы не запутались в этом нагромождении терминов: указатели на объекты, ссылки на указатели, указатели на указатели и т. д. Помните, что указатель является обычной переменной, которую можно передать по значению, по ссылке либо по указателю. Это просто неудачная запись в нотации C++ (унаследованная от C), допускающая две интерпретации интерфейса функции.

```
void read (ifstream& f, Person* person)      // считывание одной записи
{ char buf[80] ;
  Person* p = new Person;                    // выделение памяти для нового объекта
  . . .                                      // остальная часть read()
  person = p; }                              // присоединение нового объекта
```

Здесь Person* person можно интерпретировать либо как объект Person, передаваемый по указателю, либо как указатель Person (типа Person*), передаваемый по значению. Передать этот указатель по ссылке совсем не сложно. Стандартное правило C++ (описанное в главе 7 "Программирование с использованием функций C++") указывает, что для перехода от передачи по значению к передаче по ссылке необходимо только вставить амперсанд (&) между наименованием типа и именем параметра. Другие изменения не нужны, ни в теле функции, ни в синтаксисе вызова функции. Вот так он должен выглядеть.

```
void read (ifstream& f, Person* &person)        // считывание одной записи
{ char but[80] ;
  Person* p = new Person;                       // выделение памяти для нового объекта
  . . .                                         // остальная часть read()
  person = p; }                                 // присоединение нового объекта
```

Чтобы облегчить этот переход, параметру было первоначально присвоено имя Person* person, а не Person *person. Но это не играет роли. C++ не учитывает пробелы в этом случае, и можно расположить звездочку (и амперсанд) между наименованием типа и именем параметра наиболее подходящим способом.

Передача указателя по указателю не является проблемой. Необходимо быть осторожным при вызове функции, интерфейсе функции и в теле функции. В вызове функции звездочка вставляется перед именем переменной (т. е. перед именем указателя data[cnt]). Вызов функции выглядит так:

```
while (!from.eof())                     // считывание до eof
{ read(from, &data[cnt]);               // передача указателя по указателю
  cnt++; }
```

В интерфейсе функции звездочка вставляется перед наименованием Параметра, т. е. вместо Person* person следует указывать Person* *person (или Person** person).

В теле функции, и именно здесь чаще всего возникают ошибки, звездочку следует использовать перед именем параметра. Имя параметра — person, а не *person или **person. Следовательно, последний оператор в функции read() должен использовать разыменование:

```
void read (ifstream& f, Person** person)        // указатель по указателю
{ char buf[80];
  Person *p = new Person;                       // выделение памяти для нового объекта
  . . .                                         // остальная часть read()
  *person = p; }                                // приз!
```

Это совсем нетрудно, но передача указателя по ссылке намного проще передачи указателя по указателю.

До сих пор обсуждались технические детали ввода данных в массив внутри компьютера. Мы не рассматривали динамическое связывание, полиморфизм и другие вопросы. Мы считаем, что повторение материала из 6 и 7 глав по динамическому управлению памятью, файлам ввода/вывода и передаче параметров пойдет только на пользу.

Динамическое связывание становится проблемой, когда программа начинает обработку данных, которые уже находятся в памяти. Объекты разных видов требуют разной обработки, поэтому программе необходимо распознавать в каждом конкретном вызове, из какой области обрабатывается объект. Здесь пригодится поле kind класса Person. В этом простом примере "обработка данных" означает тщательный просмотр массива указателей и вывод на печать каждого объекта Person либо как преподавателя (с отображенным rank — служебным положением), либо как студента (с отображенной major — специализацией). В реальной жизни

должно быть несколько функций, которые интерпретируют по-разному различные виды объектов. В данном примере эта обработка помещена в main().

```
for (int i=0; i < cnt; i++)                    // просмотр массива указателей
{ cout <<" id: " <<data[i]->id <<endl;         // вывод на печать
                                               // идентификатора, имени
  cout <<" name: " <<data [i]->name <<endl ;
  if (data[i]->kind == 1)
    cout <<" rank: " <<data[i]->rank <<endl;       // должность на факультете
  else if (data[i]->kind == 2)
    cout <<" major: " <<data[i]->major <<endl;     // специализация студента
  cout << endl; }
```

Этот цикл является центральной частью программы: он обрабатывает список неоднородных объектов в соответствии с фактическим типом объекта. В первую очередь он безоговорочно выполняет все то, что должно осуществляться для всех типов объектов,— выводит на печать идентификатор университета и имя с соответствующим сопроводительным текстом. Чтобы обработать каждый объект в соответствии с его типом, в цикле осуществляется доступ к полю kind объекта, указывающему его тип. Он выводит на печать либо служебное положение, либо специализацию.

В листинге 15.3 приведена программа, обрабатывающая входной файл (см. рис. 15.9). Дополнительно к типу Person и функции read() программа содержит функцию main(), выполняющую роль клиентской программы. Она определяет массив указателей и файловых объектов, считывает и обрабатывает в цикле вводимые данные, а затем возвращает динамическую память. Результаты выполнения программы представлены на рис. 15.11.

```
Total records read: 4

Id: U12345678
name: Smith, John
rank: Associate Professor

Id: U12345611
name: Jones, Jan
major: Computer Science

Id: U12345689
name: Black, Jeanne
rank: Assistant Professor

Id: U12345622
name: Green, James
major; Astronomy
```

Рис. 15.11. *Вывод результатов обработки неоднородного списка объектов*

Листинг 15.3. Обработка неоднородного списка — традиционный подход

```
#include <iostream>
#include <fstream>
using namespace std;

struct Person {
       int kind;                          // 1 для преподавателей, 2 для студентов
       char id[10];                       // фиксированной длины
       char* name;                        // переменной длины
       char* rank;                        // только для преподавателей
       char* major;                       // только для студентов
   } ;

void read (ifstream& f, Person*& person)  // считывание одной записи
{  char buf[80];
   Person* p = new Person;                // выделение памяти для нового объекта
   f.getline(buf,80);                     // распознавание входящего типа
   if (strcmp(buf, "FACULTY") == 0)
      p->kind = 1;                        // 1 для преподавателей
   else if (strcmp(buf, "STUDENT") == 0)
      p->kind = 2;                        // 2 для студентов
   else
      p->kind = 0;                        // тип неизвестен
   f.getline(p->id,10);                   // считывание идентификатора
   f.getline(buf,80);                     // считывание имени
   p->name = new char[strlen(buf)+1];     // выделение памяти
```

```
    strcpy(p->name, buf);                          // копирование имени
    f.getline(buf,80);                             // считывание rank/major
    if (p->kind == 1)
    { p->rank = new char[strlen(buf)+1];           // память для rank
      strcpy(p->rank, buf); }                      // копирование rank
    else if (p->kind == 2)
    { p->major = new char[strlen(buf)+1];          // память для major
      strcpy(p->major, buf); }                     // копирование major
    person = p;                                    // присоединение к массиву
}

int main()
{
    Person* data[20]; int cnt = 0;                 // массив указателей
    ifstream from("univ.dat");                     // файл входных данных
    if (!from) { cout << " Cannot open file\n"; return 0; }
    while (!from.eof())
    { read (from, data[cnt]);                      // считывание до eof
      cnt++; }
    cout << " Total records read: " << cnt << endl << endl;
    for (int i=0; i < cnt; i++)
    { cout <<" id: " <<data[i]->id <<endl;         // вывод на печать идентификатора, имени
      cout <<" name: " <<data[i]->name <<endl;
      if (data[i]->kind == 1)
         cout <<" rank: " <<data[i]->rank <<endl;  // должность на факультете
      else if (data[i]->kind == 2)
         cout <<" major: " <<data[i]->major <<endl; // специализация студента
      cout << endl; }
    for (int j=0; j < cnt; j++)
    { delete [] data[j]->name;                     // удаление имени
      if (data[j]->kind == 1)
         delete [] data[j]->rank;                  // удаление rank/major
      else if (data[j]->kind == 1)
         delete [] data[j]->major;
      delete data[j]; }                            // удаление записи
    return 0 ;
    }
```

В этом решении нет ничего сложного или запутанного (за исключением, возможно, записи указателя). Хотя здесь и используется много ловушек языка C++ (например, структуры, указатели, динамическое управление памятью в операторах new и delete, передача параметра по ссылке, библиотечные файловые объекты), подобную программу можно написать на любом языке. В ней реализовано динамическое связывание (каждый объект обрабатывается в соответствии с его собственным типом). Однако преимущества объектно-ориентированных возможностей языка не используются (например, связывание данных и операций вместе, конструкторы и деструкторы, передача обязанностей серверам, наследование и т. д.).

Динамическое связывание: объектно-ориентированный подход

В следующем ниже варианте программы создадим три класса: Person, Faculty и Student. Все возможности, общие для обработки данных преподавателей и студентов, перейдут в базовый класс Person — поля id, name и kind. Вместо использования чисел для обозначения типа объекта (1 — для преподавателей, 2 — для

студентов), представим перечислимый тип с информативными значениями. Это особенно удобно, если имеется несколько видов объектов и могут добавляться объекты нового типа.

Элементы данных базового класса Person определены как защищенные, а не как закрытые. Производные классы Faculty и Student смогут осуществлять доступ к этим элементам данных.

```
struct Person {
public:
  enum Kind { FACULTY, STUDENT } ;
protected:
  Kind kind;                        // FACULTY или STUDENT
  char id[10];                      // данные общие для обоих типов
  char* name;                       // переменной длины
public:
  Person(const char id[], const char nm[], Kind type);
  Kind getKind() const;
  -Person(); } ;
```

Конструктор принимает три параметра для инициализации элементов данных трех объектов. Он выполняет те операции, которые в предыдущем варианте осуществлялись в функции read(): динамическое выделение памяти для имени. Деструктор выполняет операции, которые ранее были представлены в функции main(): освобождение памяти динамически распределяемой области. Это хороший пример соединения в один класс того, что в ином случае могло бы быть разделено на части и выделено для разных фрагментов программы.

Другая функция-член getKind() является вспомогательной. Это сообщение будет отправлено клиентской программе (функция read()) для вычисления объектов производных классов (Faculty и Student). В предыдущем варианте функция read() осуществляла непосредственный доступ к полю kind, создавая, таким образом, зависимость между различными частями программы. В данной структуре поле kind является защищенным, не закрытым, а класс Person должен предоставить функцию доступа для обслуживания клиентов своих производных классов.

Как можно видеть, больше внимания уделяется клиентской программе, а не производным классам. Для производных классов был разрешен прямой доступ к базовым элементам данных. Для клиентской программы предусмотрены функции доступа к серверным элементам данных.

Производные классы Faculty и Student наследуются из Person открыто. Несмотря на то, что для их определения используется зарезервированное слово struct, режим наследования определяется явно, во избежание путаницы. Если бы режим был пропущен, он был бы общедоступным по умолчанию. Общедоступный режим является наиболее естественным и удобным, он не запрещает клиентам производных классов использование возможностей базового класса. В данном случае это не важно. Базовый класс настолько мал, что рекомендуем вам использовать метод getKind().

Тем не менее использование общедоступного наследования имеет большое значение. Здесь применяется массив указателей типа Person, но предполагается установить эти указатели на объекты классов Faculty и Student. Для этого используется приведение типов. Чтобы неявное приведение стало возможным, в C++ должен быть общедоступный режим наследования.

При необходимости можно использовать и явное приведение. Но есть и другой, более важный вопрос. По существу в этой структуре предполагается использовать виртуальные функции. Они позволяют клиентской программе вызывать метод производного класса, например write(), и разрешают среде выполнения программ определять, к которому из производных классов принадлежит эта функция.

Такое поведение возможно только в том случае, если режим порождения является общедоступным.

```
struct Faculty : public Person {          // общедоступное наследование
private:
  char* rank;                             // только для преподавателей
public:
  Faculty(const char id[], const char nm[], const char r []);
  void write () const;                    // отображение записи
  -Faculty(); } ;                         // возвращение памяти динамически
                                          // распределяемой области памяти

struct Student : public Person {          // общедоступное наследование
private:
  char* major;                            // только для студентов
public:
  Student(const char id[], const char nm[], const char m[]);
  void write () const;                    // отображение записи
  -Student(); } ;                         // возвращение памяти динамически
                                          // распределяемой области памяти
```

Производные классы Faculty и Student наследуют все элементы данных своего базового класса Person. Они определяют свои собственные данные, конкретные для каждого вида Person (rank или major).

Конструкторы производных классов Faculty и Student принимают параметры, необходимые для инициализации всех своих полей, независимо от того, определены ли они в производном классе или унаследованы из базового класса Person. Задача конструктора производного класса состоит в передаче данных конструктору базового класса в списке инициализации. Как можно установить из интерфейса конструктора Person, к ним относятся спецификации вида создаваемых объектов, Faculty или Student. Для многих программистов это означает, что список параметров конструкторов производных классов должен включать данные для инициализации базовой части (три параметра) и данные для инициализации производной части (major для Student, rank для Faculty).

```
Faculty(const char id[], const char nm[], Kind k, const char r[])
    : Person(id,nm,k)                              // список инициализации
  { rank = new char[strlen(r)+1];
    if (rank == 0) { cout << "Out of memory\n"; exit(0); }
    strcpy(rank,r); }
```

Это типичный пример делегирования обязанности в клиентскую часть производного класса. Клиентская программа (функция read()) создает объекты Faculty примерно так:

```
person = new Faculty(id,name,FACULTY,buf); }     // объектом является Faculty
```

Но это обман, клиентская программа уже объявила о создании объекта Faculty. Зачем делать бесполезную работу по передаче параметров, объявляя это как Faculty? Подобная обязанность должна быть передана объекту Faculty. Он знает, что он является Faculty, и должен сказать об этом своей части Person, не втягивая клиента read() в цикл совместной работы.

```
Faculty(const char id[], const char nm[], const char r[])
      : Person (id, nm, FACULTY)                  // именно в этом суть ООП
  { rank = new char[strlen(r)+1];
    if (rank == 0) { cout << "Out of memory\n"; exit(0); }
    strcpy(rank,r); }
```

Теперь клиентская программа должна выполнить более простую работу.

```
person = new Faculty(id,name,buf); }              // объектом является Faculty
```

Обратите внимание, что прототипы конструктора в спецификациях Faculty и Student содержат три параметра, а не четыре. В этом состоит суть объектно-ориентированного программирования: правильное распределение обязанностей между взаимодействующими классами.

Деструкторы производных классов возвращают память, выделенную в динамически распределяемой области памяти конструкторами (rank для Faculty, major для Student).

Функция-член write() реализуется в обоих производных классах. Они подобны, поэтому их имена могут быть одинаковыми. Необходимо реализовать write() в основном классе. Но алгоритмы для различных видов Person не идентичны. Именно поэтому для каждого класса существует отдельная функция, но они используют одно и то же имя. По существу эти функции называются полиморфными.

Функции write() представляют часть функциональных возможностей функции main() из предыдущего варианта. Поскольку каждый производный класс Faculty и Student знает свои возможности, вам не надо выяснять вид целевого объекта.

```
void Faculty::write () const          // отображение записи
{ cout << " id: "<< id << endl;       // вывод на печать идентификатора, имени
  cout << " name: "<< name" << endl ;
  cout << " rank: "<< rank <<endl <<endl;-}      // только для преподавателей

void Student::write () const          // отображение записи
{ cout << " id: << "id << endl;       // вывод на печать идентификатора, имени
  cout << " name: "<< name << endl;
  cout << " major: "<< major <<endl <<endl; }    // только для студента
```

Глобальная функция read() представляет модернизированную модификацию функции из предыдущего варианта программы. Она считывает данные из входного файла в локальные массивы, а затем проверяет массив kind[], чтобы видеть, объект какого типа требуется построить. Если она сообщает "FACULTY", то функция read() создает новый объект Faculty с помощью оператора new. Если она сообщает "STUDENT", read() задает новый объект Student с помощью оператора new. В любом случае данные отправляются конструктору класса как параметры.

```
void read (ifstream& f, . . . ?? person)        // какой у него тип?
{ char kind[8], id[10] , name[80], buf[80];
  f.getline(kind,80);                           // распознавание входного типа
  f.getline(id.10);                             // считывание идентификатора
  f.getline(name,80);                           // считывание имени
  f.getline(buf,80);                            // rank или major?
  if (strcmp(kind, "FACULTY") == 0)
  { person = new Faculty(id,name,buf); }        // объект - Faculty
  else if (strcmp(kind, "STUDENT") == 0)
  { person = new Student(id,name,buf); }        // объект - Student
  else
  { cout << " Corrupted data: unknown type\n", exit(0); } }
```

Тип второго параметра этой функции должен быть указателем. В противном случае он не сможет принимать значение, возвращаемое оператором new. Оно должно передаваться по ссылке, а не значением. Иначе новый объект будет обозначаться только локальным указателем person, а не фактическим аргументом. Кроме того, новый объект будет недоступен из клиентской программы. Это не может быть указатель Faculty. Такой указатель не может ссылаться на объект Student. Он не может быть указателем Student, поскольку не может указывать на объект Faculty.

Итак, это не может быть ни указатель Faculty, ни указатель Student. Какой же должен быть тип указателя, способный ссылаться на объекты различных классов? Вспомните, если различные классы не связаны наследованием, то отсутствует указатель, который может указывать на объекты этих классов и выполнять любую работу. Кроме того, если разные классы связываются наследованием, то указатель базового класса может указывать на объекты любого производного класса — A, B или иного. "Старший брат" может указывать на все, что захочет, поскольку класс назначения находится в пределах иерархической структуры наследования.

Следовательно, это должен быть указатель Person. Внутри функции read() объекты разных производных классов создаются и подключаются к указателю базового класса.

```
void read (ifstream& f, Person*& person)       // считывание одной записи
{ char kind[8], id[10], name[80], buf[80];
  f.getline(kind,80);                          // распознавание входного типа
  f.getline(id,10);                            // считывание идентификатора
  f.getline(name,80);                          // считывание имени
  f.getline(buf,80);                           // rank или major?
  if (strcmp(kind, "FACULTY") == 0)
  { person = new Faculty(id,name,buf); }       // объект - Faculty
  else if (strcmp(kind, "STUDENT") == 0)
  { person = new Student(id,name, buf); }      // объект - Student
  else
  { cout << " Corrupted data: unknown type\n" ; exit(0); } }
```

Функция read() вызывается из main() так же, как и в предыдущем варианте. Отличие состоит в том, что компоненты массива data[] типа Person* теперь указывают на объекты других производных классов (Faculty или Student).

```
int main()
{ cout << endl << endl;
  Person* data[20]; int cnt = 0;                  // массив указателей
  ifstream from("univ.dat");                      // файл входных данных
  if (!from) { cout << " Cannot open file\n"; return 0; }
  while (!from.eof())
  { read(from, data[cnt]);                         // считывание до eof
    cnt++; }
  . . . }                                          // остальная часть main()
```

Однако базовый указатель не может вызывать операции, которые определены в производных классах. До тех пор, пока базовый указатель ссылается на производный объект, всегда существует способ сообщить компилятору то, что нам известно: базовый указатель указывает на производный объект. Для этого следует воспользоваться приведением к производному классу.

Этот процесс принятия решения хотелось бы инкапсулировать в функцию, например write(). Подобные функции для выбора решения должны проектироваться для каждой операции, которые выполняются по-разному для различных типов подобных объектов. Трудно выбрать тип параметра для этой функции. Общая структура данной функции:

```
void write ( . . . ?? p)          // отображение записи
{ switch (p.getKind()) {          // получение типа объекта
    case Person::FACULTY:
         . . .;  break;           // выполнить как для Faculty
    case Person::STUDENT:
         . . .;  break; } }       // выполнить как для Student
```

Тип параметра этой функции должен воспринимать два типа фактических аргументов — объекты Faculty и Student. Если тип объекта — Faculty, то эта функция будет вызывать только Faculty::write(). Если тип параметра Student, функция будет вызывать только Student::write().

Именно здесь следует воспользоваться материалом из предыдущего раздела о преобразованиях классов. Можно воспользоваться параметром типа Person, поскольку и объекты Faculty, и объекты Student могут копироваться в объект Person. (Вспомним, что объект производного класса располагает достаточными данными для инициализации объекта базового класса.)

```
void write (Person p)                  // отображение записи
{ switch (p.getKind()) {               // получение типа объекта
    case Person::FACULTY:
         . . .;  break;                // выполнить как для Faculty
    case Person::STUDENT:
         . . .;  break; } }            // выполнить как для Student
```

Недостатком такого решения является то, что он передает параметр по значению, а это не слишком хорошая практика, когда объекты управляют своей памятью динамически. Кроме того, тело функции остается на уровне объекта Person и отсутствует способ для преобразования базового объекта обратно в объект производного класса. Исходные данные отбрасываются и их невозможно восстановить. Даже если добавить конструктор производного класса, который преобразует базовый объект в объект производного класса (см. листинг 15.2), этого будет недостаточно. Конструктор сможет лишь установить значения по умолчанию полей для производного класса. Нам, однако, требуются исходные значения служебного положения преподавателя или специализации студента.

Вы не можете использовать значения базового объекта в качестве параметра функции, но ничто не мешает вам применить базовый указатель как параметр функции. Указатель производного класса (который выполняет все операции производного класса) можно преобразовать в указатель базового класса — это безопасное преобразование, поэтому приведение типов не требуется. Данные не отбрасываются и возможно обратное преобразование в указатель производного класса. (Однако это преобразование не является безопасным и, следовательно, требуется приведение типа.)

```
void write (Person* p)                 // отображение записи
{ switch (p->getKind()) {              // получение типа объекта
    case Person::FACULTY:
         . . .;  break;                // выполнить как для Faculty
    case Person::STUDENT:
         . . .;  break; } }            // выполнить как для Student
```

Во время такого преобразования нельзя выполнять операции, определенные для производного класса. Слабый базовый указатель может лишь достичь функций, которые определены в базовом классе. Но преимущество этого решения состоит в том, что этот базовый указатель все еще указывает на объект производного класса. В операторе выбора функция write() выясняет, указывает ли фактический параметр на объект Faculty или на объект Student. Остается только вызвать либо метод write() из класса Faculty, либо метод write() из класса Student.

```
void write (const Person* p)           // отображение записи
{ switch (p->getKind()) {              // получение типа объекта
    case Person::FACULTY:
         p->write();  break;           // выполнить как для Faculty
    case Person::STUDENT:
         p->write();  break; } }       // выполнить как для Student
```

Указатель p является указателем базового класса, поэтому он может добраться только до методов базового класса. Следовательно, вызовы write() в обеих ветвях оператора выбора либо будут достигать write() из базового класса (если он есть в классе Person), либо приведут к синтаксической ошибке (если в классе Person метод write() отсутствует).

Функция write() уже знает, на объект какого типа указывает ее параметр-указатель. Компилятор знает только, что это указатель на класс Person. Значит, функция write() должна сообщить компилятору о том, что она знает. Компилятор должен выполнить приведение базового указателя либо к классу Faculty (первый оператор выбора), либо к классу Student (второй оператор выбора).

```
void write (const Person* p)                    // отображение записи
{ switch (p->getKind()) {                       // получение типа объекта
    case Person::FACULTY:
        ((Faculty*)p)->write();   break;        // выполнить как для Faculty
    case Person: .-STUDENT:
        ((Student*)p)->write();   break;        // выполнить как для Student
} }
```

Это приведение выглядит внушительным и устрашающим. Но оно осуществляет приведение указателя p класса Person* в указатель типа Faculty* или в указатель типа Student*. Скобки используются потому, что операнд выбора в виде стрелки имеет более высокий приоритет, чем операнды приведения. Если опустить скобки и использовать, например, (Faculty*)p->write(), компилятор решит, что нужно преобразовать значение, возвращаемое в результате вызова write(), а не указатель p. Сохраните здесь эти скобки.

Эта функция write() будет вызываться в цикле, получая в качестве фактических аргументов указатели Person, которые указывают либо на Faculty, либо на Student объекты.

```
for (int i=0; i < cnt; i++)
  { write(data[i]); }                   // отображение данных
```

Полная программа представлена в листинге 15.4.

Листинг 15.4. Обработка неоднородного списка — объектно-ориентированный подход

```
#include <iostream>
#include <fstream>
using namespace std;

struct Person {
public:
      enum Kind { FACULTY, STUDENT } ;
protected:
      Kind kind;                                    // FACULTY или STUDENT
      char id[10] ;                                 // данные общие для обоих типов
      char* name;                                   // переменная длина

public:
      Person(const char id[], const char nm[], Kind type)
      { strcpy(Person::id,id);                      // копирование идентификатора
        name = new char[strlen(nm)+1];              // выделение памяти для имени
        if (name ==0) { cout << "Out of memory\n" ; exit(0); }
        strcpy(name,nm);                            // копирование имени
        kind = type; }                              // помните его тип
```

```
    Kind getKind() const
      { return kind; }                              // доступ к типу Person

       ~Person()
       { delete [] name; }          // возврат памяти динамически распределяемой области памяти
} ;

struct Faculty : public Person {
private:
       char* rank;                                  // только для преподавателей

public:
       Faculty(const char id[], const char nm[], const char r[])
          : Person(id,nm,FACULTY)                   // список инициализации
       { rank = new char[strlen(r)+1];
         if (rank == 0) { cout << "Out of memory\n"; exit(0); }
         strcpy(rank,r); }

   void write () const                              // отображение записи
   { cout << " id: " << id << endl;                 // вывод на печать идентификатора, имени
     cout << " name: " << name << endl;
     cout << " rank: " << rank <<endl <<endl; }     // только для преподавателей

       ~Faculty()
       { delete [] rank; }          // возврат памяти динамически распределяемой области памяти
} ;

struct Student : public Person {
private:
       char* major;                                 // для студента

public:
       Student(const char id[], const char nm[], const char m[])
          : Person id,nm,STUDENT)                   // инициализация списка
       { major = new char[strlen(m)+1];
         if (major == 0) { cout << "Out of memory\n"; exit(0); }
         strcpy(major,m); }

   void write () const                              // отображение записи
   { cout << " id: " << id << endl;                 // вывод на печать идентификатора, имени
     cout << " name: " << name << endl;
     cout << " major: " << major <<endl <<endl; }   // только для студента

       ~Student()
       { delete [] major; }         // возврат памяти динамически распределяемой области памяти
} ;

void read (if stream& f, Person*& person)           // считывание одной записи
{  char kind[8], id[10], name[80], buf[80];
   f.getline(kind,80);                              // распознавание входного типа
   f.getline(id,10);                                // считывание идентификатора
   f.getline(name,80);                              // считывание имени
   f.getline(buf,80);                               // rank или major?
   if (strcmp(kind, "FACULTY") == 0)
   { person = new Faculty(id,name,buf); }           // объект - Faculty
   else if (strcmp(kind, "STUDENT") == 0)
   { person = new Student(id,name,buf); }           // объект - Student
   else
   { cout << " Corrupted data: unknown type\n" exit(0); }
}
```

```
void write (const Person* p)                          // отображение записи
{ switch (p->getKind()) {                             // получение типа объекта
     case Person::FACULTY:
          ((Faculty*)p)->write();   break;            // выполнить как для Faculty
     case Person::STUDENT:
          ((Student*)p)->write();   break;            // выполнить как для Student
} }

int main()
{
  cout << endl << endl;
  Person* data[20]; int cnt = 0;                      // массив указателей of pointers
  ifstream from("univ.dat");                          // файл входных данных
  if (! from) { cout << " Cannot open file\n"; return 0; }
  while (!from.eof())
  { read(from, data[cnt]);                            // считывание до eof
    cnt++; }
  cout << " Total records read: " << cnt << endl << endl;
  for (int i=0; i < cnt; i++)
  { write(data[i]); }                                 // отображение данных
  for (int j=0; j < cnt; j++)
  { delete data[j]; }                                 // удаление записи
  return 0;
  }
```

Это решение намного элегантнее, чем предыдущее. Данные и операции связаны, действия перенесены в серверные классы, разделение на части связанного кода исключено. Как и при любом объектно-ориентированном подходе, исходная программа слишком длинная. В ином случае программа выполняет то, что и программа в листинге 15.3. Ее вывод соответствует выводу программы в листинге 15.3 (см. рис. 15.11).

На следующем этапе следует исключить проверки типа объекта, которые выполняются функцией `write()`. Вместо тестирования типа объекта-цели, приведения указателя обратно к этому типу, а затем вызова функции соответствующего производного класса, потребуем от компилятора осуществить все это. Компилятор должен сгенерировать код объекта, который тестирует тип объекта, выполняет приведение и вызывает соответствующий метод. Для этого рекомендуем использовать зарезервированное слово `virtual` в назначении функций членов базового класса.

Динамическое связывание: виртуальные функции

Зарезервированное слово `virtual` является синтаксическим маневром. Оно создает свойство разрешения типа во время исполнения для сообщения, отправленного объекту производного типа. Для использования этого свойства в базовом классе и в каждом производном классе реализуется функция с одинаковым именем.

В качестве примера рассмотрим эти средства, реализуя метод `write()` для базового класса `Person` и для производных классов `Faculty` и `Student`. Вы сможете написать глобальную функцию `write()` следующим образом:

```
void write (const Person* p)          // отображение записи
{ p->write(); }                       // разве это не красиво?
```

Для связывания в процессе компиляции это просто означает вызов метода `write()`, определенного в классе `Person`. Для связывания во время выполнения программа,

сгенерированная компилятором, проанализирует тип объекта, на который указывает базовый указатель p, определит, к какому типу должен относиться вызываемый метод, и вызовет метод write() из этого типа. В зависимости от объекта, обозначенного указателем, будет вызван либо метод Faculty, либо метод Student.

Чтобы метод работал, необходимо выполнить несколько ограничений. Виртуальная функция, принадлежащая производному классу, должна вызываться только через базовый указатель или базовую ссылку. Связывания во время выполнения не произойдет, если сообщение посылается базовому объекту или объекту производного класса. В каждом случае используется алгоритм для статического связывания. Сообщение вызывается из того класса, каким является тип объекта.

Например, значение x.write() зависит от типа, к которому принадлежит объект x. Этот тип определяется в процессе компиляции, а не во время выполнения.

Виртуальная функция не может быть статической. Она не может вызваться из оператора области действия класса, но должна вызываться через базовый указатель (ссылку), которая ссылается на объект производного класса.

Режим наследования для порождения должен быть общедоступным и не может быть защищенным или закрытым. Неявное приведение допускается только для общедоступных порождений.

Функция с этим же именем определяется как виртуальная в базовом классе иерархии наследования. В каждом производном классе должна быть реализована функция с таким же именем, что и базовая виртуальная функция. При переопределении функции в производном классе она должна совпадать по имени, сигнатуре и типу возвращаемого значения с виртуальной функцией базового класса.

Если имя функции в производном классе другое, это не является ошибкой. Однако такая функция не может вызываться с использованием связывания во время исполнения. Динамическое связывание использует вызов той же самой функции, но с другой ее интерпретацией.

Если в производном классе другая сигнатура, то производный метод скрывает базовый метод и разрушает механизм виртуальной функции. Если производные классы определяют пустую функцию write() без параметров, а базовый класс обозначает пустую функцию write(int), то при использовании динамического связывания отсутствует способ для вызова функций производного класса. В этом случае p->write() вызовет функцию, которая принадлежит к классу указателя p. Если она существует, вы сможете ее вызвать. В противном случае имейте в виду, что допущена синтаксическая ошибка.

Если тип результата виртуальных функций в производных классах другой, это синтаксическая ошибка, даже если сигнатура функций одна и та же.

Зарезервированное слово virtual появляется только в спецификации базового класса. В определении функции базового класса, а также в спецификации производного класса его не требуется повторять.

Если иерархия включает классы более чем двух уровней, виртуальные функции можно определять на любом уровне иерархии. Совсем не обязательно реализовать определенную функцию, например, на самом верхнем или на более низком уровне иерархии наследования. Она может наследоваться косвенно.

Если все ограничения удовлетворены, не надо определять поле kind в базовом классе и метод, который возвращает значение поля kind. Для преобразования программы в листинге 15.3 в программу с виртуальными функциями требуется определить функцию в классе Person. У функции должен быть тип результата void и параметры должны отсутствовать.

```
struct Person {
protected:
  char id[10];                    // Kind отсутствует
  char* name;
```

```
public:
  Person(const char id[], const char nm[]);       // Kind отсутствует
  virtual void write () const;                    // const - часть сигнатуры
  ~Person(); } ;
```

В результате производным классам не требуется передавать базовому классу информацию поля kind.

```
struct Faculty : public Person {
private:
       char* rank;                                // только для преподавателей
public;
       Faculty(const char id[], const char nm[], const char r[])
            : Person (id, nm)                     // FACULTY отсутствует
       { rank = new char[strlen(r)+1];
            if (rank == 0) { cout << "Out of memory\n"; exit(0); }
            strcpy(rank,r); }
  void write () const                             // теперь является виртуальной
  { cout << " id: " << id << endl;    // вывод на печать идентификатора, имени
    cout << " name: " << name << endl;
    cout << " rank: " << rank <<endl <<endl; }    // только для преподавателей
       ~Faculty()
       { delete [] rank; } } ;                    // возврат памяти динамически
                                                  // распределяемой области памяти
```

Что более важно, в клиентской программе не требуется проверять тип объекта. В листинге 15.5 приведена программа из листинга 15.4, использующая виртуальную функцию write() для исключения из клиентской программы анализа подтипа. Вывод программы такой же, что и для предыдущего варианта (см. рис. 15.11).

Листинг 15.5. Обработка неоднородного списка с использованием виртуальных функций

```
#include <iostream>
#include <fstream>
using namespace std;

struct Person {
protected:
       char id[10];                                // данные общие для обоих типов
       char* name;                                 // переменной длины

public:
       Person(const char id[], const char nm[])    // типа Kind
       { strcpy(Person::id,id);                    // копирование идентификатора
         name = new char[strlen(nm)+1];            // выделение пространства для объекта
         if (name == 0) { cout << "Out of memory\n"; exit(0); }
         strcpy(name,nm);                          // копирование имени
       }

       virtual void write() const                  // работы не много
       { }

       v~Person()              // возврат памяти динамически распределяемой области памяти
       { delete[] name; }                          // для объекта Person
} ;

struct Faculty : public Person {
private:
       char* rank;                                 // только для преподавателей
```

```
public:
        Faculty(const char id[], const char nm[], const char r[])
          : Person(id,nm)                              // список инициализации
        { rank = new char[strlen(r)+1];
          if (rank == 0) { cout << "Out of memory\n"; exit(0); }
          strcpy(rank,r); }

   void write () const                                 // отображение записи
   { cout << " id: " << id << endl;                    // вывод на печать идентификатора, имени
     cout << " name: " << name << endl;
     cout << " rank: "<< rank <<endl <<endl; }         // только для преподавателей

        ~Faculty()
        { delete[] rank; }          // возврат памяти динамически распределяемой области памяти
} ;

struct Student : public Person {
private:
        char* major;                                   // только для студента

public:
        Student(const char id[], const char nm[], const char m[])
          : Person(id, nm)                             // список инициализации
        { major = new char[strlen(m)+1];
          if (major == 0) { cout << "Out of memory\n"; exit(0); }
          strcpy(major,m); }

   void write() const                                  // отображение записи
   { cout << " id: " << id << endl;                    // вывод на печать идентификатора, имени
     cout << " name: " << name << endl ;
     cout << " major: " << major <<endl <<endl; }      // только для студента

        ~Student()
        { delete[] major; }         // возврат памяти динамически распределяемой области памяти
   } ;

   void read (if stream& f, Person*& person)           // считывание одной записи
   { char kind[8], id[10], name[80], buf[80] ;
     f.getline(kind,80);                               // распознавание входного типа
     f.getline(id,10);                                 // чтение идентификатора
     f.getline(name,80);                               // чтение имени
     f.getline(buf,80);                                // rank или major?
     if (strcmp(kind, "FACULTY") == 0)
     { person = new Faculty(id,name,buf); }            // объект - Faculty
     else if (strcmp(kind, "STUDENT") == 0)
     { person = new Student(id,name,buf); }            // объект - Student
     else
     { cout << "C orrupted data: unknown type\n"; exit(0); }
   }

void write (const Person* p)                           // отображение записи
{ p->write(); }                                        // Faculty или Student

int main()
{  cout << endl << endl;
   Person* data[20]; int cnt = 0;                      // массив указателей
   ifstream from("univ.dat");                          // файл входных данных
   if (!from) { cout << " Cannot open file\n"; return 0; }
   while (!from.eof())
   { read(from, data[cnt]);                            // считывание до eof
     cnt++; }
```

```
cout << " Total records read: " << cnt << endl << endl;
for (int i=0; i < cnt; i++)
{ write(data[i]); }                              // отображение данных
for (int j=0; j < cnt; j++)
{ delete data[j]; }                              // удаление записи
return 0 ;
}
```

Полиморфизм (интерпретация сообщений объектам во время исполнения) основывается на допустимости неявного приведения типов объектов из производного в базовый класс. Базовый указатель (в примере — Person) может указывать на производный объект (Faculty или Student) без применения явного приведения.

```
Person *p, *pf, *ps;                             // указатели типа Person
p = new Person("U12345678", "Smith");
pf = new Faculty("U12345689", "Black", "Assistant Professor");
ps = new Student("U12345622", "Green", "Astronomy");
```

Приведение является оптимальным методом. Оно может использоваться для привлечения внимания лиц, осуществляющих сопровождение, к преобразованиям типов указателей.

```
ps = (Person*) new Student("U12345622", "Green", "Astronomy");
```

Виртуальные функции не требуют обязательного приведения типа сообщения обратно к типу объекта, на который указывает производный указатель. Обратите внимание, что указатели на производные объекты не могут ссылаться на базовые объекты без явного приведения типа. Производный указатель также не может вызвать базовый метод без явного приведения типов.

```
Student* s = (Student*) ps;          // приведение типов обязательно
s->write();                          // указатель производного класса
```

В виртуальных функциях использование базовых указателей приводит в результате к вызовам функций-членов производного класса.

```
ps->write();                        // указатель базового класса
```

Однако все эти улучшения влияют только на внешний вид клиентской программы. По своей сути программа, представленная в листинге 15.5, выполняет то же самое, что и программа в листинге 15.4. Поле kind происходит из класса Person, но фактически находится здесь. Причем оно доступно программе, сгенерированной компилятором, а не исходной программе, написанной программистом. Оператор выбора происходит из клиентской программы, но он также находится здесь. Он реализован программой, сгенерированной компилятором, а не исходной программой, написанной программистом.

Программа в листинге 15.4 явно выделяет дополнительную память для анализа типа объектов Person и тратит время на принятие решения, какую функцию write() вызвать. Программа в листинге 15.5 выделяет такую же дополнительную память и тратит дополнительное время.

Некоторые программисты, разрабатывающие системы управления реального времени, говорят, что виртуальные функции неэкономны. Это несправедливо. Для полиморфного алгоритма требуется выделить дополнительную память и затратить время. А реализован ли он явно, как в листинге 15.4, или с виртуальными функциями, как в листинге 15.5, не имеет большого значения.

Динамическое и статическое связывание

Динамическое связывание предлагает программисту новый и увлекательный способ структурирования обрабатывающих алгоритмов. Можно создать семейство связанных производных классов в общем открытом базовом классе, снабдить каждый производный класса функцией, которая выполняет обработку способом, конкретным для этого производного класса, удостовериться, что у всех функций одинаковое имя и интерфейс, а затем вызвать эту функцию через указатель базового класса. В результате вы увидите, что вызываемая функция не зависит от типа объекта, обозначенного указателем.

Но динамическое связывание не уменьшает значения традиционного статического связывания. В большинстве случаев программирования на C++ вызываемый метод зависит от типа указателя, ссылающегося на объект, а не от типа объекта, на который указывает указатель. Появляются дополнительные проблемы.

Для статического связывания при анализе вызова функции необходимо рассмотреть тип цели сообщения и сигнатуру вызываемого метода. При возможном использовании динамического связывания необходимо принять во внимание несколько дополнительных факторов.

Во-первых, необходимо знать, является ли цель сообщения объектом или указателем (ссылкой). Если это объект, возможно только статистическое связывание и требуется учитывать сигнатуру метода для проверки того, что вызов функции правильный. Если целью сообщения является указатель или ссылка, динамическое связывание возможно.

Во-вторых, следует определить, к какой точке иерархии наследования принадлежит указатель. Если указатель базового типа, то динамическое связывание возможно — оно зависит от типа объекта, на который он указывает, и от того, как определена функция. Если указатель относится к одному из производных типов, возможно только статическое связывание. Однако результат вызова также зависит от типа указываемого объекта и от способа определения функции.

Следовательно, нужно принять во внимание только два фактора: тип указываемого объекта и способ определения функции. Объект может быть базового типа (динамическое связывание невозможно) и одного из производных типов (динамическое связывание возможно только в случае, если на объект указывает базовый указатель). Функция может определяться либо в базовом, либо в производном классе. Также функция может задаваться как в базовом классе, так и производном классе. В этом случае следует различать функции, переопределенные в производном классе с той же сигнатурой, что и в базовом классе, и функции с другой сигнатурой. Динамическое связывание могут поддерживать только те функции, которые переопределены с той же сигнатурой. Другие функции допускают лишь статическое связывание, и не все из них могут быть вызваны для заданной комбинации типа указателя и типа объекта. Следует различать четыре вида функций-членов:

- Функции, определенные в базовом классе и наследованные в производном классе без переопределения
- Функции, определенные в производном классе без прототипа в базовом классе
- Функции, определенные в базовом классе и переопределенные в производном классе с тем же самым именем и с той же или другой сигнатурой
- Функции, определенные в базовом классе и переопределенные в производном классе как виртуальные с тем же самым именем и с той же сигнатурой

Для базового указателя, обозначающего базовый объект, могут вызываться только методы, определенные в базовом классе, независимо от того, унаследованы они в производных классах без изменений или были там переопределены.

Попытка вызвать функцию, определенную в производном классе без прототипа в базовом классе, является синтаксической ошибкой. Попытка вызвать функцию, переопределенную в производном классе, несерьезна — функция, определенная в базовом классе, вызывается везде.

Для производного указателя, ссылающегося на производный объект (того же класса) базовые функции недоступны, за исключением тех, которые определены в базовом классе и наследуются без изменений. Этот указатель может вызывать методы, которые добавляются в производный класс и переопределяются в производном классе. Функции, переопределенные в производном классе, вызываются статически независимо от того, как они определены — с той же сигнатурой или с другой, как виртуальные функции или нет.

Обратите внимание, что базовые функции, переопределенные в производном классе, недоступны для указателя производного класса, ссылающегося на объект производного класса. Они скрыты соответствующими функциями производного класса. Попытка достичь базовой функции приведет к статическому вызову функции, определенной в производном классе (виртуальном или не виртуальном), если сигнатуры совпадают, или к синтаксической ошибке, если нет.

Указатель базового класса, ссылающийся на объект производного класса, может вызывать базовые методы, унаследованные (но не переопределенные) производным классом. Он не может достичь методов, определенных в производном классе и не имеющих прототипа в базовом классе. Если производный класс переопределяет базовый метод как не виртуальную функцию (либо с той же самой, либо с другой сигнатурой), этот производный метод также не может быть вызван через базовый указатель. Вместо этого соответствующий базовый метод будет вызываться статически. Если производный класс переопределяет базовый метод как виртуальную функцию, то через указатель базового класса вызывается метод производного класса, а не базового класса. Это единственный случай, когда возможно динамическое связывание.

Указатель производного класса, указывающий на базовый объект, является аномалией. Он может вызывать методы, определенные в базовом классе и унаследованные в базовом классе без переопределения. Он не может вызвать методы базового класса, переопределенные в производном классе, поскольку они скрыты от этого указателя. Он не может вызвать методы производного класса, которые переопределяют методы базового класса (как виртуальные, так и не виртуальные с той же самой или с другой сигнатурой), поскольку они не поддерживаются базовым объектом, а при попытке сделать это возникает ошибка.

Описание основывается на двух принципах:

- Производный указатель, ссылающийся на производный объект, может достичь методов, определенных в производном классе, и методов, унаследованных из базового класса без изменений. Методы, переопределенные в производном классе, скрывают методы, определенные в базовом классе, от указателя производного класса.
- Базовый указатель, указывающий на производный объект, может достичь этих методов, определенных в базовом классе. Но есть одно исключение. Если функция переопределена в производном классе как виртуальная, то базовый указатель, используя динамическое связывание, вызывает функцию производного класса, а не базового класса.

Это очень просто, но для выполнения такой операции может потребоваться время. Эти правила представлены в графическом виде (см. рис. 15.12 и таблицу 15.1).

На рис. 15.12 представлены указатели базового класса (узкие прямоугольники) и указатели производного класса (более широкие прямоугольники из двух частей), которые указывают на объекты базового класса (часть, показанная пунктирной линией, представляет собой пропущенную производную часть) и производного класса (левая часть представляет базовую часть, правая часть производную часть).

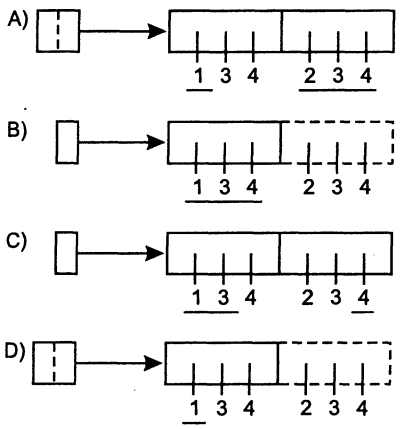


Рис. 15.12.
Статическое и динамическое связывание для указателей базового и производного класса

Вертикальные линии внутри каждой части обозначают функции-члены четырех типов. Тип 1 определяется в базовом классе и наследуется в производном классе без изменений. Тип 2 добавляется к производному классу, он не имеет прототипа в базовом классе. Тип 3 определяется в базовом классе и переопределяется в производном классе с тем же самым именем. Тип 4 задается в базовом классе (как виртуальный) и переопределяется в производном классе с тем же именем и той же сигнатурой.

Методы, которые могут вызываться через указатель, подчеркнуты. В случае A функции типа 3 и 4, определенные в базовом классе, скрываются функциями, определенными в производном классе. В случае B доступны только функции, определенные в базовом классе. В случае C разрешаются только функции, определенные в базовом классе, но функции, переопределенные в производном классе как виртуальные, скрывают свои прототипы базового класса и могут вызываться динамически. В случае D могут вызываться только функции, определенные в базовом классе и не переопределенные в производном классе.

В таблице 15.1 перечислены эти же правила. Столбцы показывают типы объектов и типы указателей, которые ссылаются на объекты. Строки описывают различные виды функций-членов.

Таблица 15.1

Краткое перечисление правил статического и динамического связывания

Виды функций-членов	Базовые указатели		Производные указатели	
	Базовый объект	*Производный объект*	*Базовый объект*	*Производный объект*
Функции, определенные в классе `Base`				
Наследованы в классе `Derived` без изменений	доступны	доступны	доступны	доступны
Переопределены в `Derived` (не виртуальные)	доступны	доступны	недоступны	скрыты
Переопределены в `Derived` (виртуальные)	доступны	скрыты	недоступны	скрыты
Функции, определенные в классе `Derived`				
Определены только в классе `Derived`	синтаксическая ошибка	синтаксическая ошибка	аварийная ситуация	доступны
Переопределены в `Derived` (не виртуальные)	недоступны	недоступны	аварийная ситуация	доступны
Переопределены в `Derived` (виртуальные)	недоступны	динамическое связывание	аварийная ситуация	доступны

Чисто виртуальные функции

Базовые виртуальные функции могут не выполнять никаких действий, потому что они не имеют значения в рамках приложения. Их задача — определить наследование как стандарт для всех производных классов. Именно поэтому виртуальные функции представляются в первую очередь.

Например, метод `write()` в классе `Person` ничего не содержит. В нем нет кода. Обратите внимание, что он никогда не вызывается. Все вызовы метода `write()` в клиентской программе (глобальная функция `write()`) разрешаются либо в классе `Faculty`, либо в методе `write()` класса `Student`.

Фактически класс Person является обобщением. В приложении отсутствуют объекты Person. Все объекты создаются оператором new в глобальной функции read() и относятся либо к классу Student, либо к Faculty. Описание проблемы в начале этой части свидетельствует о том, что существуют два вида записей — одна для студентов и одна для преподавателей. Класс Person первоначально был введен в приложение как абстракция, которая объединяет характеристики объектов профессорско-преподавательского состава и объектов студентов в один обобщенный класс (листинг 15.3). Позднее он использовался для определения иерархии производных классов (листинг 15.4). В последней версии программы (листинг 15.4) класс Person применялся для определения интерфейса виртуальной функции write().

В реальной жизни класс Person может быть очень полезным. В нем могут быть не только идентификатор и наименование университета, но и дата рождения, адрес, номер телефона и другие характеристики, обычные для объектов Faculty и Student. Кроме того, класс Person может определять такие многочисленные методы, как изменение имени, адреса или номера телефона, извлечение идентификатора университета и других данных, обычных для объектов Faculty и Student. Производные классы могут наследовать все эти полезные функции. Клиенты производных классов используют подобные функции, посылая сообщения, определенные в классе Person, объектам классов Faculty и Student. Снова не было сказано, что класс Person — бесполезен. Отмечалось, что объекты класса Person являются бесполезными для этого приложения. Приложению требуются только объекты классов, производных от Person. Помните об этом.

Проектировщик класса Person знает, что приложение не создает объекты класса и что для объектов класса нет задания для выполнения. Было бы прекрасно передать эту информацию программисту клиентской части и лицам, осуществляющим сопровождение, не через комментарии, а в самой программе. Язык C++ позволяет определять базовый класс таким образом, что попытка создания объекта этого типа будет недопустимой и приведет к синтаксической ошибке.

Язык C++ делает это возможным через использование чистых виртуальных функций и абстрактных классов. Не совсем ясно, почему два термина — "чистый" и "абстрактный" — используются для описания одной и той же идеи. Чистой виртуальной функцией является виртуальная функция, которая не должны вызываться (подобно write() в классе Person). Если программа пытается ее вызвать, возникает синтаксическая ошибка. Абстрактный класс — это класс с не менее чем одной чистой виртуальной функцией. Не допускается создание объектов подобного класса. Если программа пытается создать объект этого класса либо динамически, либо в стеке, появляется синтаксическая ошибка.

Для чистых виртуальных функций и абстрактных классов в C++ отсутствуют ключевые слова. Вместо них чистая виртуальная функция распознается (компилятором, клиентской программой и лицом, осуществляющим сопровождение) функцией-членом, которая в объявлении "инициализируется" нулем. Приведем класс Person, функция-член write() которой определяется как чистая виртуальная функция.

```
struct Person {                                    // абстрактный класс
protected:
    char id[10];                                   // данные, общие для обоих типов
    char* name;
public:
    Person(const char id[], const char nm[]);
    virtual void write () const = 0;               // чистая виртуальная функция
    ~Person() ;
} ;
```

Оператор присваивания не обозначает присваивание. Это еще один пример придания символу дополнительного значения в следующем контексте. Добавление другого ключевого слова, такого как чистый или абстрактный, возможно, было бы лучшим решением.

Чистая виртуальная функция не имеет реализации. Фактически предоставление реализации чистой виртуальной функции (или вызов функции) является синтаксической ошибкой. Именно присутствие виртуальных функций делает класс абстрактным (или частичным) классом.

Абстрактный класс должен иметь хотя бы один производный класс. В нем не должно быть объектов с приписанными значениями. Если производный класс реализует эту функцию, он становится регулярным классом. В противном случае он становится абстрактным классом. Создание объекта этого производного класса не допускается, и такой класс должен иметь не менее одного производного класса.

Производные классы реализуют чистые виртуальные функции так же, как и обычные виртуальные функции. Это означает, что производный класс должен использовать то же самое имя, сигнатуру и возвращаемый тип, которые будут применяться чистой виртуальной функцией. Режим порождения должен быть общедоступным. Ниже приводится пример класса Faculty, который реализует виртуальную функцию write(). Это регулярный неабстрактный класс.

```
struct Faculty : public Person {            // регулярный класс
private:
    char* rank;                             // только для преподавателей
public:
    Faculty(const char id[], const char nm[], const char r[]);
    void write() const;                     // регулярная виртуальная функция
    ~Faculty();
} ;
```

Этот же производный класс использовался в листинге 15.5. Рассматривая регулярный неабстрактный класс, можно видеть, является ли он производным от абстрактного класса или от регулярного класса. Пользователю класса Faculty совершенно все равно, как реализован базовый класс Person, в той мере, в которой клиентская программа не пытается приписать значения объектам абстрактного класса.

Для регулярного класса с виртуальными функциями клиентская программа может создать объекты, отправить им сообщения и, если требуется, использовать полиморфизм.

Абстрактный класс является классом языка C++. Он может включать элементы набора данных и регулярные, не чистые функции, даже виртуальные функции.

Если класс наследует виртуальную функцию как чистую виртуальную функцию, не определяя ее тело, этот производный класс является абстрактным. Никакие объекты данного класса не могут быть созданы. Если для клиентской программы нужны объекты этого класса, можно использовать незаполненное тело подобной функции. Класс становится регулярным, неабстрактным классом, и можно создать его объект.

```
class Base {                                // абстрактный класс
public:
  virtual void member() = 0;                // чистая виртуальная функция
  . . . . } ;                               // оставшаяся часть класса Base

class Derived : public Base {               // регулярный класс
public:
  void member()                             // виртуальная функция
    { }                                     // пустое тело: поор
  . . . . } ;                               // оставшаяся часть класса Derived
```

Класс Base является абстрактным классом. Его объекты не могут быть созданы. Класс Derived представляет собой регулярный класс. Его объектам могут быть присвоены значения в стеке (как именованные переменные) или в динамически распределяемой области памяти (как неименованные переменные). Функция member() в классе Base является чистой виртуальной функцией. Вызвать ее невозможно. Функция member() в классе Derived является регулярной виртуальной функцией. Однако ее вызов приводит в результате к отсутствию операций.

```
Base *b; Derived *d;      // указатели Base и Derived
b = new Base;             // синтаксическая ошибка, абстрактный класс
d = new Derived;          // OK, регулярный класс, объект динамически
                          // распределяемой области памяти
b = new Derived;          // OK, неявное преобразование указателя
d->member();              // OK, связывание на этапе компиляции,
                          // холостая команда
b->member();              // OK, связывание во время выполнения
d->Base::member();        // ошибка компоновщика: реализация отсутствует
```

Переопределение с другой сигнатурой делает функцию в производном классе не виртуальной. Здесь класс Derived1 является классом, который наследуется из абстрактного класса Base, но не переопределяет чистую функцию member() без параметров. Вместо этого он задает функцию member(int) с одним параметром.

```
class Derived1 : public Base {     // также абстрактный класс
  public:
    void member(int)               // не виртуальная функция
      { }                          // пустое тело: холостая команда
  . . . . } ;                      // оставшаяся часть класса Derived
```

Это означает, что класс Derived1 является абстрактным классом. Создание его объектов является синтаксической ошибкой. Поскольку этот класс не используется как базовый класс для порождения других классов, он бесполезен.

```
class Derived2 : public Derived1 {     // регулярный класс
  public:
    void member()                      // виртуальная функция
      { }                              // пустое тело: холостая команда
  . . . . } ;                          // оставшаяся часть класса Derived
```

Класс Derived2 наследуется из класса Derived1. Он реализует виртуальную функцию-член member(), следовательно, допускается создание объектов этого класса. Его объекты могут отвечать на сообщение member() как со связыванием на этапе выполнения, так и со статическим связыванием. Объекты этого класса не могут отвечать на сообщение member(int), потому что функция скрывается функцией-членом member(), определенной в классе Derived2.

```
Derived2 *d2 = new Derived2;     // OK, регулярный класс, объект динамически
                                 // распределяемой области памяти
d2->member();                    // OK, статическое связывание
b = new Derived2;                // OK для виртуальной функции
b->member();                     // OK, динамическое связывание
b->member(0);                    // синтаксическая ошибка
d2->member(0);                   // неверное количество параметров
```

Обратите внимание, что указатель b базового класса при обозначении объекта производного класса может вызвать:

- Не чистые функции-члены (виртуальные или не виртуальные), определенные в базовом классе
- Виртуальные функции, заданные в производном классе

Он вызывает только виртуальные функции-члены, определенные в производном классе. Это указатель ближнего действия. Он использует виртуальную функцию для расширения ее области видения до производной части объекта, на который он указывает. В противном случае она сможет видеть только базовую часть производного объекта. Указатель производного класса используется для осуществления доступа как к базовой части, так и к производной части производного объекта.

Виртуальные функции: деструкторы

При вызове оператора удаления называется деструктор и объект уничтожается. Вы вызываете деструктор, определенный в классе указателей, ссылающихся на объект, или деструктор, определенный в классе, к которому принадлежит объект, на который указывает указатель.

Когда указатель и объект принадлежат к одному классу, деструктор принадлежит этому классу.

```
Derived2 *d2 = new Derived2 ;      // OK, регулярный класс, объект динамически
                                   // распределяемой области памяти
d2->member();                      // OK, статическое связывание
b = new Derived2;                  // OK для виртуальных функций
b->member();                       // OK, динамическое связывание
delete d2;                         // деструктор класса Derived2
delete b;                          // ??
```

Деструкторы C++ являются регулярными, не виртуальными функциями-членами. Когда используется оператор delete, компилятор находит определение операнда указателя, затем класса, к которому принадлежит указатель, и вызывает деструктор. Все это происходит во время компиляции. Компилятор не обращает внимания на класс объекта, на который указывает указатель.

Когда указатель и объект принадлежат одному классу, проблемы отсутствуют. При выполнении кода деструктора динамическая память и другие ресурсы, выделенные объекту, возвращаются. Когда указатель производного класса ссылается на базовый объект, это делать не требуется. Большой и мощный указатель производного класса потребует от небольшого базового класса выполнить то, что он не может.

```
Person p; Faculty f;               // базовый и производный указатели
p = new Person("U12345678", "Smith");
f = p;                             // синтаксическая ошибка: следует избегать
f = (Faculty*)p;                   // именно это нужно выполнить
delete f;                          // деструктор Faculty
```

В этом примере деструктор Faculty вызывается в объекте Person. Оператор удаления delete вызывается для компонента набора данных rank, т. е. не в объекте. Результаты не определены.

Когда базовый указатель ссылается на объект производного класса, вызывается деструктор базового класса. Если динамическая память обрабатывается в базовом, а не в производном классе, то проблем не возникает. Память, выделенная из динамически распределяемой области памяти, будет возвращена ей базовым деструктором. Если управление памятью динамически распределяемой области памяти осуществляется в производном классе, она не будет возвращена базовым деструктором. В результате происходит "утечка" памяти.

```
Person *p; Faculty* f;                  // базовый и производный указатели
f = new Faculty("U12345689", "Black", "Assistant Professor");
p = f;                                  // или p = (Person*) f;
delete p;                               // "утечка" памяти
```

В данном примере оператор delete вызывает деструктор Person, который удаляет динамическую память, выделенную для имени. Деструктор Faculty не вызывается, а память, выделенная в динамически распределяемой области памяти для rank, не возвращается.

В листинге 15.5 клиентская программа использует цикл для перехода к массиву базовых указателей и удаляет каждый объект, выделенный динамически в начале выполнения. Для каждого объекта в структуре данных выполняется деструктор Person.

```
for (int j=0; j < cnt; j++)
{ delete data[j]; }            // возвращение памяти, выделенной для Person,
                               // динамически распределяемой области памяти
```

Для Faculty и Student их память возвращается полностью. Оператор delete удаляет объект независимо от его типа. Проблема связана с памятью динамически распределяемой области памяти, выделенной для объектов производного класса (см. рис. 15.10). Деструктор Person удаляет память динамически распределяемой области памяти, выделенную для имени, но не память динамически распределяемой области памяти, выделенную для rank и major. Когда производный объект уничтожается через базовый указатель, вызывается только базовый деструктор.

Для решения подобной проблемы C++ предлагает объявить деструктор Base виртуальным. Условно говоря, деструктор каждого производного класса также станет виртуальным. Когда оператор delete применяется к базовому указателю, деструктор класса назначения вызывается полиморфным способом (а затем, если имеется, деструктор базового класса).

```
struct Person {                                  // абстрактный класс
protected:
  char id [10];                                  // данные, общие для обоих типов
  char* name;
public:
  Person(const char id[], const char nm[]);
  virtual void write() const = 0;                // чистая виртуальная функция
  virtual ~Person() ;                            // в этом весь фокус!
} ;

struct Faculty : public Person {                 // регулярный класс
private:
  char* rank;                                    // только для преподавателей
public :
  Faculty(const char id[], const char nm[], const char r[]);
  void write () const;                           // регулярная виртуальная функция
  ~Faculty() ;                                   // теперь также виртуальный
} ;
```

Это решение громоздкое. Первое, что следует помнить о виртуальных функциях, это то, что и в базовом классе, и во всех производных классах используется одно и то же имя. С деструкторами это не так, поскольку каждый из них имеет то же имя, что и имя класса. Следовательно, деструкторы нарушают правила для виртуальных функций. Это подобно конструкторам, и в C++ виртуальные конструкторы отсутствуют.

Однако с "утечками" памяти опасно мириться. Именно поэтому C++ поддерживает виртуальные деструкторы.

Множественное наследование: несколько базовых классов

В C++ производный класс может иметь более одного базового класса. При простом наследовании классы расположены следующим образом: на вершине иерархии находится базовый класс, ниже — производный класс.

При множественном наследовании иерархия классов может стать скорее графом, чем деревом, как в простом наследовании. Вопросы, связанные с множественным наследованием, трудны для понимания.

Множественное наследование — это методика, позволяющая облегчить создание серверной программы. В отличие от простого наследования множественное разрешает проектировщику серверного класса смешивать характеристики различных классов в одном классе.

Рассмотрим простой пример. Предположим, что класс B1 предоставляет клиентам открытый сервис f1(), а класс B2 — открытый сервис f2(). Это почти то же самое, что требуется клиентской программе. Дополнительно для этих двух сервисов клиентской программы необходим открытый сервис f3(). Одной из возможных методик для обслуживания клиента должно быть объединение характеристик классов B1 и B2 в одном классе с помощью множественного наследования.

```
class B1
  { public:
       void f1();          // открытый сервис f1()
    ... };                 // оставшаяся часть класса B1

class B2
  { public:
       void f2();          // открытый сервис f2()
    ... };                 // оставшаяся часть класса B2
```

При открытом наследовании от классов B1 и B2 класс Derived способен обеспечить своих клиентов объединенными сервисами, предоставляемыми каждому из базовых классов (в данном случае методы f1() и f2()). Это означает, что для обеспечения своих клиентов всеми тремя сервисами (f1(), f2() и f3()) проектировщик класса Derived должен реализовать только одну функцию f3().

```
Class Derived : public B1, public B2     // два базовых класса
  { public:                              // f1(), f2() наследуются
       void f3();                        // f3() добавляется к сервисам
    ... } ;                              // оставшаяся часть класса B2
```

Теперь клиентская программа может присвоить значения объектам Derived и отправить им сообщения, которые они наследовали из обоих базовых классов, и сообщения, которые добавляются классом Derived.

```
Derived d;                 // присваивание значения объекту Derived
d.f1(); d.f2();            // унаследованные сервисы (B1, B2)
d.f3();                    // сервисы добавляются в класс Derived
```

Класс Derived предоставляет клиентам возможности всех базовых классов, плюс их собственные данные и поведение.

Первоначально язык C++ не располагал множественным наследованием. Но Страуструп, разработчик C++, жаловался, что программисты "требовали множественное наследование", и теперь оно в C++ есть.

Множественное наследование хорошо подходит для настройки существующих библиотек классов, например, для добавления или переопределения членов имеющихся классов. Производные классы представляют комбинацию базовых классов,

а не уточнение отдельного базового класса. Каждый родительский класс вносит свои элементы в производный класс. Производный класс является объединением базовых возможностей.

Примерами использования множественного наследования являются графические объекты, счета NOW и классы `iostream` в стандартной библиотеке C++.

Для графического пакета классы `Shape` и `Position` использовались как базовые классы для создания класса `Object`. Объекты класса `Object` объединили свойства объектов `Shape` и `Position`. Это пример неразумного применения множественного наследования. Графические объекты являются фигурами, но трудно утверждать, что они представляют собой положения. Скорее можно говорить о том, что графический объект располагается в каком-то месте.

Для счетов NOW классы представляют собой сберегательные и текущие счета. Это лучший пример использования множественного наследования. Счет NOW действительно объединяет свойства сберегательных и текущих счетов. По нему выплачиваются проценты и разрешается выписывать чеки. Однако если расспросить служащего банка, можно узнать, что бывают исключительные ситуации, когда счет NOW отличается как от сберегательного, так и от текущего счета. Это означает, что преимущества легкого слияния основных характеристик компенсируются недостатками подавления свойств, которые не соответствуют друг другу.

Для библиотеки C++ класса `iostream` имеет смысл использовать множественное наследование для слияния характеристик классов входных и выходных потоков. Полученные в результате классы `iostream` поддерживают как операции ввода, так и операции вывода, а в производных классах ничего не требуется подавлять.

Обратите внимание на то, что C++ не устанавливает ограничения на количество базовых классов, которые могут участвовать в формировании производного класса. Все приведенные примеры включают только два базовых класса. Сложно придумать примеры множественного наследования с тремя или четырьмя базовыми классами так, чтобы они имели смысл и не запутывали пользователя. Почему два лучше, чем три или четыре? Кажется, что примеры множественного наследования с двумя базовыми классами так же трудно понять.

Именно поэтому рекомендуется использовать множественное наследование осторожно. Имейте в виду, что существуют способы поддержки клиентской программы без использования множественного наследования.

Множественное наследование: правила доступа

При множественном наследовании производный класс наследует элементы данных всех базовых классов и все функции-члены этих классов. Область памяти, которую занимает объект производного класса, представляет собой сумму пространства, занимаемого в памяти объектами базовых классов (возможно, с учетом выравнивания).

Правила доступа для множественного наследования те же, что и для простого наследования. Доступ к методам класса `Derived` могут осуществлять общедоступные и защищенные члены всех других базовых классов без каких-либо ограничений. Вы не имеете доступ к закрытым членам базовых классов.

Связи наследования могут быть общедоступными, защищенными или закрытыми. В любом случае все элементы данных и функции-члены базовых классов наследуются производным классом. Однако в зависимости от способа порождения могут изменяться права доступа.

Способы порождения для множественного наследования те же, что и для простого наследования. При открытом порождении каждый закрытый, защищенный и общедоступный член базового класса имеет те же права доступа в объектах производного класса, что и в базовом объекте. Это наиболее естественный режим наследования.

При защищенном порождении защищенные и общедоступные базовые члены остаются защищенными и общедоступными в производном классе, но общедоступные базовые члены (даты и функции) становятся в производном классе защищенными. Так как производный класс имеет полный доступ к защищенным базовым компонентам, защищенное наследование не оказывает влияния на права доступа производного класса. Подобно простому наследованию, оно влияет на права доступа клиентской программы. В этом случае клиентская программа теряет право использовать открытые базовые сервисы. Производный класс должен предусматривать адекватные сервисы для клиентов, не делая открытые базовые сервисы доступными для клиентской программы.

При закрытом наследовании все базовые члены становятся закрытыми в производном классе. Подобно простому наследованию, способ порождения по умолчанию является закрытым. Режим порождения должен определяться для каждого базового класса отдельно.

Рассмотрим два одинаковых базовых класса Base B1 и B2.

```
class B1
  { public:
       void f1();          // открытый сервис f1()
    ... } ;                // оставшаяся часть класса B1

class B2
  { public:
       void f2();          // открытый сервис f2()
    . . . } ;              // оставшаяся часть класса B2
```

Объединим их характеристики в порожденном классе Derived и добавим еще одну функцию-член в порожденном классе.

```
class Derived : public B1, B2      // два базовых класса
  { public:                        // f1(), f2() наследуются
       void f3();                  // f3() добавляется к сервисам
    ... };                         // оставшаяся часть класса Derived
```

Затем клиентская программа может определять и использовать объекты класса Derived.

```
Derived d;              // присваивание значения объекту Derived
d.f1();                 // унаследован из B1
d.f2();                 // синтаксическая ошибка: f2() закрытый
d.f3();                 // сервисы добавляются в класс Derived
```

Это еще одно проявление различий между зарезервированным словом public, используемым для описания прав доступа и для описания способа порождения. В правах доступа область видимости зарезервированного слова public включает столько членов класса, сколько требуется, до тех пор, пока не будет обнаружено другое зарезервированное слово для определения прав доступа. В способе порождения область видимости зарезервированного слова public включает только один идентификатор.

В приведенном выше примере существует только один класс B1, из которого класс Derived наследуется открыто. Для класса B2 используется способ порождения по умолчанию (закрытый). Метод f2() становится закрытым в классе Derived и будет не доступен клиентской программе.

Преобразования классов

Правила преобразования для множественного наследования и простого наследования подобны. Если базовый класс наследуется из общедоступного класса, то объекты производного класса могут быть неявно преобразованы в объекты этого базового класса. *Для такого преобразования оператор явного приведения не требуется.*

Объект производного класса располагает всеми возможностями, данными и функциями объектов базовых классов. Преобразование из производного объекта в базовый объект не может привести в результате к потере возможностей. Это может произойти в случае, если способ порождения не является общедоступным.

```
B1 b1; B2 b2; Derived d;
B1 = d; b2 = d;                  // OK; дополнительные возможности отвергаются
d = b1; d = b2;                  // ошибка: несогласованное состояние объекта
```

Преобразование из базового класса в производный класс не допускается. Базовый объект содержит только часть данных и возможностей, которыми располагает производный объект, а пропущенные возможности не могут быть добавлены. Такое преобразование не безопасно.

Подобные же правила применяются к указателям и ссылкам. Указатель (ссылка) базового класса может безопасно указывать на объект производного класса. Объект производного класса может выполнять все то же, что и базовый указатель. Это безопасно. Однако базовый указатель может вызвать любую часть возможностей производного объекта.

```
B1 *p1; B2 *p2; Derived *d;
P1 = new Derived; p2 = new Derived;      // OK: безопасно
d = new B1; d = new B2;                  // синтаксические ошибки
d = p1; d = p2;                          // синтаксические ошибки
d = (Derived*) p1;                       // OK: явное приведение
```

Указатель производного класса не должен указывать на базовый объект (третья строка примера). В базовом объекте отсутствуют многие возможности, имеющиеся у производного объекта, которые доступны через указатель производного класса. Чтобы избежать ошибок во время выполнения, компилятор объявляет этот код синтаксической ошибкой.

Подобным образом базовый указатель (который, по-видимому, указывает на базовый объект) не может быть скопирован в указатель производного класса (четвертая строка примера). Это не безопасно. Производный указатель может потребовать сервисы, которые базовый объект не в состоянии выполнить, а компилятор не может за этим проследить. Следовательно, манипулирование указателем также рассматривается как синтаксическая ошибка.

Как поступить, если известно, что базовый указатель ссылается на объект производного класса, а не на базовый объект? Укажите компилятору, что вам известно, что вы делаете с помощью приведения типов.

Эти же правила применяются к передаче параметра. Если функция ожидает указатель (или ссылку) на один из базовых классов, то безопаснее вызвать эту функцию, передавая ей адрес производного объекта.

```
void foo1 (B1 *b1)          // производные объекты содержат
                            // дополнительные свойства
  { b1->f1(); }

void foo2 (B2 *b2)          // производные объекты содержат
                            // дополнительные свойства
  { b2->f2(); }
```

```
void foo(Derived *d)        // базовые объекты не могут выполнить это
  { d->f3(); }

B1 *b1 = new Derived; B2 *b2 = new Derived;
Derived d;
Foo1(&d); foo2(&d);         // оба - OK: безопасное преобразование
foo(b1); foo(b2);           // синтаксические ошибки: опасное преобразование
foo((Derived*)b1); foo((Derived*)b2);        // передается на свой риск
```

В последнем примере функции foo1() и foo2() могут принять объекты Derived как фактические аргументы, поскольку внутри этих функций параметры отвечают только на базовые сообщения (f1() и f2()), а производные объекты — на эти сервисы. Функция foo() не может принимать базовые указатели, потому что внутри нее их параметр должен отвечать на сообщения производного класса f3(), а базовые объекты этого не могут сделать. С другой стороны, указатели b1 и b2 ссылаются на объекты класса Derived, которые выполняют такое задание. Чтобы сообщить это компилятору, последняя строка программы, приведенной выше, выполняет явное приведение указателя Base в указатель Derived.

В закрытом или защищенном режиме наследования не допускаются неявные преобразования из объектов производного класса в объекты базового класса. Даже в этом "безопасном" случае требуется явное приведение в клиентской программе. Преобразование из любого базового класса в производный класс требует явного приведения для любого вида множественного наследования.

Множественное наследование: конструкторы и деструкторы

Производный класс отвечает за состояние своих компонентов, унаследованных от базовых классов. Как и в простом наследовании, конструкторы базового класса вызываются, когда строится объект производного класса.

Механизм передачи параметров конструкторам базового класса подобен механизму для простого наследования. Должен использоваться список инициализации элементов. В следующем примере базовый класс B1 содержит один элемент данных, базовый класс B2 — другой элемент данных, а производный класс — еще один элемент данных (динамически выделенный массив символов). Класс Derived должен обеспечить для конструктора три параметра, чтобы он мог передать данные своим компонентам B1 и B2 и собственному элементу данных.

```
class B1 {
  int m1;
public:
  B1(int) ;
  void f1(); . . . . };

class B2 {
  double m2;
public:
  B2(double);
  void f2();. . . . };

class Derived: public B1, public B2 {
  char* t;
public:
  Derived(const char*, double, int);
  ~Derived();
  void f3(); ... };
```

Если список инициализации элементов не предусматривается, то вызывается конструктор Base по умолчанию. Если базовые классы не имеют в виду конструкторы по умолчанию, то это синтаксическая ошибка.

В списке инициализации элементов конструктор класса Derived вызывает базовые конструкторы, используя имена классов B1 и B2 в последовательности вызовов конструкторов, разделенных запятыми. Имена параметров для базовых конструкторов обычно поступают из списков параметров конструктора Derived.

```
Derived::Derived(const char *s, double d, int i) : B1(i),B2(d)
  { if ((t = new char[strlen(s)+1] ) == NULL)
       { cout << "\nOut of memory\n";  exit(1); }
    strcpy(t,s); }
```

Все конструкторы базового класса вызываются до вызова конструкторов производного класса. Располагаются они в том порядке, в котором базовые классы перечислены в объявлении производного класса.

Подобно простому наследованию, элементы наборов данных производного класса могут инициализироваться либо в теле конструктора производного класса, либо в списке инициализации элементов.

При уничтожении объекта производного класса (динамически или при выходе из области видимости) вначале вызывается деструктор производного класса, а затем деструкторы базового класса в порядке, обратном вызову конструкторов.

Множественное наследование: неоднозначность

Использование множественного наследования может привести к конфликтам имен. Если производный класс содержит элемент данных или функцию с тем же именем, что и один из базовых классов, то сервис базового класса скрывается именем, определенным в производном классе.

В следующем примере класс Derived содержит элемент данных x с тем же именем, что и элемент данных в базовом классе B1. Кроме того, класс Derived имеет функцию-член f2() с тем же именем, что и функция-член в базовом классе B2.

```
class B1 {
protected:
  int x;                        // скрыто Derived::x
public:
  void f1(); ...};

class B2 {
public:
  void f2(); ...};              // скрыто Derived::f2()

class Derived: public B1, public B2 {
protected:
  float x;                      // скрывает B1::x
public:
  void f2();                    // скрывает B2::f2()
  void f3()
    { x = 0; }... };            // используется Derived::x
```

В этом примере объект класса Derived содержит два элемента данных x; элемент данных, наследованный из B1, скрывается в Derived; функция-член f2(), наследованная из B2, скрывается добавленной функцией f2().

Как клиентская программа, так и программа класса Derived может подменять правила области видимости, используя явный оператор области действия.

```
void Derived::f3()
  { B1::x = 0; }              // игнорируя Derived::x
```

```
Derived d;
d.f2();                    // Der::f2();
d.B2::f2();                // B2::f2();
```

Конфликты между именем производного класса и именами базового класса случаются не очень часто. Обычно проектировщик производного класса имеет возможность просмотреть структуру базовых классов и избежать конфликтов.

Имена элементов базового класса также могут конфликтовать. С этим бороться намного сложнее, потому что базовые классы часто разрабатываются независимо друг от друга. Существует небольшая возможность скоординировать их разработку, чтобы избежать конфликтов имен.

Если в двух базовых классах совпадают имена элементов данных или функций-членов, то объект производного класса содержит обе копии. Язык не предоставляет предварительно определенные правила предшествования для доступа к данным и функциям. Если необходимо, неоднозначность разрешается использованием явной квалификации. Оператор области видимости должен применяться как клиентом, так и производным классом.

В следующем примере оба класса включают открытую функцию-член с именем `f1()`. Это означает, что клиентская программа не может использовать ни один из них, если только она не обеспечивает явного указания, какой использовать.

```
class B1 {
public:
  void f1(); ... } ;

class B2 {
public:
  void f1(); ... } ;

class Derived : public B1, public B2 {
public:
  void f3(); ... } ;

Derived d;
d.f1();                                  // двусмысленное сообщение
d.B1::f1(); d.B2::f1(); d.f3();          // OK
```

Этот метод исключения неоднозначности бросает вызов принципам объектно-ориентированного программирования. Проблема разрешается добавлением больших обязанностей клиентской программе, а не серверному классу. Убедитесь, что можете узнать этот вид структур, и избегайте их.

Лучше сделать так, чтобы класс `Derived` изолировал клиентскую программу от неоднозначности имен функции-члена.

```
class Derived: public B1, public B2 {
public:
  void f1() { B1: : f1(); }              // однострочники
  void f2() { B2 : : f1(); }
  void f3(); . . . } ;

Derived d;
d.f1(); d.f2(); d.f3();                  // клиентская часть изолирована
```

Это решение намного лучше. Присутствует серверная программа (класс `Derived`), которая берет на себя часть обязанностей. Клиентская программа изолирована от проблемы. Когда клиентская программа использует класс `Derived` как сервер, она должна знать только то, как вызвать сервисы `f1()`, `f2()` и `f3()`, чтобы выполнить задание.

Если два или более базовых классов содержат элемент данных с одинаковым именем, то объект производного класса включает обе копии. В результате вы столкнетесь с неоднозначностью.

```
class B1 {
protected:
  int m;
public:
  B1(int);
  void f(); ... };

class B2 {
protected:
  double m;
public:
  B2(double);
  void f(); ... };

class Derived : public B1, public B2 {
  char* t;
public:
  Derived (char*,double,int);
  void f3() { cout << "m=" << m << endl; }        // двусмысленное выражение
  ... };
```

Конфликты между именами элементов данных должны разрешаться производным классом, чтобы избежать неоднозначности и защитить клиентскую программу. Используйте оператор области действия.

```
void Derived::f3()
{ cout << "m=" << B1::m << endl; }          // двусмысленность отсутствует
```

Множественное наследование: ориентированный граф

Это наиболее хитрая форма неоднозначности, возникающая, когда базовый класс наследуется более чем из одного класса. Как правило, C++ против этого, а класс может явно появиться только один раз в списке происхождения для производного класса.

```
class B {
public:
    int m; . . . .} ;

class Derived: public B, public B          // синтаксическая ошибка
  { .... } ;
```

В этом примере класс объявляется синтаксической ошибкой. Однако такой же класс может появляться несколько раз в иерархии наследования. Разные базовые классы могут иметь общие скобки. Подобные скобки появляются несколько раз в порождениях, и их данные в производных классах будут содержать несколько копий.

```
class B1 : public B {            // класс B приводится выше
  protected:
    int mem;
  public:
    void f1(); ... };
```

```
class B2 : public B {                          // класс B приводится выше
  protected:
    int mem;
  public:
    void f2(); ... };

class Derived : public B1, public B2 {         // унаследовано из B дважды
  public:
    void f3(); . . .};
```

В этой структуре класс Derived содержит два элемента наборов данных с именем mem, унаследованным из разных базовых классов. Имена у них одинаковые, но они указывают на разные положения в памяти. Их роли в программе также отличаются: они происходят из разных классов. Эту проблему не следует переносить на клиента.

Ситуация с элементом данных m намного хуже. Каждый объект класса Derived располагает двумя экземплярами этого элемента данных. Один унаследован через класс B1, а другой через класс B2. Пространство, которое требуется для нескольких экземпляров одного и того же базового элемента данных, тратится напрасно. Эти два элемента данных также функционально одинаковы — они происходят из одного и того же класса, но один из них обслуживает части B1 класса Derived, а второй — части B2 класса Derived.

В языке C++ предлагается интересное решение этой задачи. Программисту предоставляется возможность явно указать, что использование двух (или более) копий этих же данных и функций нежелательно. Хотелось бы, чтобы это был случай по умолчанию. Это оговаривается путем определения базовых классов виртуальными базовыми классами. Зарезервированное слово virtual модифицирует объявления производных классов, которые позже используются во множественном наследовании.

```
class B {                                      // общий базовый класс
  int m;
public:
  void f(); ... } ;

class B1 : virtual public B {                  // виртуальный базовый класс
  protected:
    int mem;
  public:
    void f1(); ... };

class B2 : virtual public B {                  // виртуальный базовый класс
  protected:
    int mem;
  public:
    void f2(); ... };

class Derived : public B1, public B2 {         // работает как волшебник
  public:
    void f3(); ... };
```

Теперь класс Derived содержит только одну копию данных и функций, наследованных из класса B. Обратите внимание, что в классе Derived именно проектировщик его базовых классов B1 и B2 должен определить эти классы как виртуальные. Это поставит под сомнение принцип, что базовые классы не знают своих производных классов и лишь производные классы уверены в своих базовых классах.

Не следует путать зарезервированное слово virtual, используемое в данном контексте, с зарезервированным словом virtual, применяемым для виртуальных функций. Они совершенно разные. Было бы прекрасно иметь два разных зарезервированных слова. Возможно, лучше, если бы множественное наследование отсутствовало.

Полезно ли множественное наследование

Трудно однозначно ответить на этот вопрос, однако по-видимому сложность структуры с множественным наследованием перевешивает преимущества его использования.

Если требуется спроектировать класс, который предоставляет своим клиентам объединение сервисов других классов, используйте составление или составление с наследованием.

Рассмотрим первый пример наследования, который уже обсуждался в начале раздела. Цель этого проекта состоит в обеспечении клиентской программы возможностью вызывать функции f1(), f2() и f3(). Функции f1() и f2() уже реализованы в классах B1 и B2. Требуется реализовать функцию f3().

```
class B1
  { public:
       void f1();              // общедоступный сервис f1()
    ... } ;                    // оставшаяся часть класса B1

class B2
  { public:
       void f2();              // открытый сервис f2()
    ... } ;                    // оставшаяся часть класса B2
```

При использовании множественного наследования требуемая функция f3() реализуется в новом классе Derived.

```
class Derived : public B1, public B2      // два базовых класса
  { public:                               // f1(), f2() наследуются
       void f3();                         // f3() добавляется к сервисам
    ... };                                // оставшаяся часть класса Derived
```

Вместо этого можно создать функцию f1(), наследующую класс Derived из класса B1. Чтобы предоставить клиентам класса Derived функцию f2(), поле класса B2 следует сделать элементом класса Derived.

```
class Derived : public B1 {       // простое наследование
  B2 b2;                          // композиция класса
public:
  void f2() { b2.f2() ; }         // однострочник
  void f3(); ... };
```

Теперь клиентская программа может приписать значение объектам Derived и отправить им сообщения точно так же, как и в случае множественного наследования.

```
Derived d;                  // присваивание значения объекту Derived
d.f1();                     // унаследованные сервисы (B1)
d.f2();                     // передано из B2 через Derived
d.f3();                     // добавлено в класс Derived
```

Данная клиентская программа не должна рассматриваться как способ проектирования класса Derived. Он предоставляет требуемые сервисы, и это все, что требуется,— без сложностей множественного наследования.

Итоги

В этой главе рассмотрены примеры расширенного использования наследования. Все они касались некоторых общих возможностей базовых и производных классов. В каких-то случаях объекты одного класса могут использоваться вместо объектов другого класса.

Показано, что использование объекта производного класса в случае, когда ожидается объект базового класса, всегда безопасно. Такое преобразование безопасно, но не очень интересно. Эти объекты должны будут выполнить только то, что может сделать базовый объект, а объект производного класса способен на большее.

Помните, что указатель базового класса можно использовать, когда ожидается указатель производного класса. То есть можно указывать объекты производного класса, используя указатели базового класса.

В языках программирования это всегда было проблемой. Все совокупности объектов, которые поддерживаются современными языками, являются однородными. Массивы C++ не могут содержать компоненты различных классов. Связанные списки C++ не могут использовать узлы разных типов. И только наследование позволяет применять совокупности объектов разных классов. Такие классы не являются совершенно разными. Неоднородные списки не содержат объекты произвольных классов, но могут включать объекты классов, связанные наследованием.

При обработке неоднородной совокупности объектов (связанных наследованием) объектам в совокупности отправляются четыре типа сообщений.

- Сообщения, на которые может ответить каждый объект в совокупности объектов. Методы, определенные в базовом классе иерархии наследования, которые не перезаписываются в производных классах.
- Сообщения, на которые могут ответить только некоторые объекты в совокупности объектов. Методы, определенные в производных классах иерархии наследования, за исключением сообщений с теми же именами в базовом классе.
- Сообщения, на которые могут ответить объекты с именами всех типов в совокупности объектов, но определенные в базовых и производных классах как невиртуальные функции (с тем же самым или с другим интерфейсом).
- Сообщения, на которые могут ответить все виды объектов в совокупности объектов, определенные как виртуальные функции, использующие один и тот же интерфейс как в базовом, так и в производных классах.

Для доступа к первому типу сообщения используйте указатель базового класса. Когда доступ к объекту осуществляется из совокупности объектов, не требуется никаких преобразований.

Отправить второй вид сообщений можно с помощью указателей производного класса. Когда объект берется из совокупности объектов, базовый указатель должен преобразовываться в указатели того класса, к которому принадлежит объект. Только тогда будут доступны сообщения второго вида. Это преобразование не является безопасным, и необходимо хорошо представлять себе, что происходит, потому что компилятор не в состоянии защитить вас.

Третий вид сообщений также требует преобразования, если объект должен отвечать на сообщение, определенное в производном классе. Сообщение базового класса скрывается сообщением производного класса.

Четвертый вид сообщения не требует преобразования. Даже если эти сообщения направляются с использованием указателя базового класса, они интерпретируются средой выполнения программ в соответствии с типом объекта, на который указывает указатель (динамическое связывание). Структура, использующая виртуальные функции, инкапсулирует алгоритмы. Они выполняются по-разному для разных видов объектов в функциях с теми же именами.

При использовании виртуальных функций увеличиваются затраты памяти и снижается производительность. Каждый объект того класса, который использует виртуальные функции, содержит скрытый элемент данных. Он определяет вид объектов или указатель, ссылающийся на таблицу с адресами доступных виртуальных функций. Каждый раз при вызове виртуальной функции этот указатель используется для поиска требуемого объектного кода. Время выполнения подобной операции увеличивается.

Рассматривалось множественное наследование. Это сложный вопрос. Рекомендуем реже использовать множественное наследование.

Виртуальные функции популярны в программировании на языке C++. Помните, однако, что механизм виртуальной функции является "хрупким". Необходимо применять общедоступное наследование. Обязательно использование тех же самых имен во всех классах своей иерархии наследования. Требуется использовать тот же список параметров, возвращаемых значений и даже модификаторов констант. При самом незначительном несоответствии программа вызовет совершенно другую функцию просто потому, что у нее то же самое имя. Иначе говоря, используйте виртуальные функции там, где обработку разного вида связанных объектов можно приемлемо описать с помощью одного и того же имени функции.

Глава 16

Расширенное использование перегрузки операций

Темы данной главы

- ✔ Перегрузка операций: краткий обзор
- ✔ Унарные операции
- ✔ Операции, возвращающие компонент массива по индексу, и операции вызова функции
- ✔ Операции ввода/вывода
- ✔ Итоги

Перегруженные операции языка C++ обсуждались в главе 10 (числовые классы) и главе 11 (нечисловые классы). В этой главе рассматриваются более экзотические варианты использования перегрузки операций в языке C++. Для некоторых программистов любая перегрузка операции сама по себе является достаточно эксцентричной.

Расширенные перегруженные операции пишутся не часто. Однако расширенные операции являются важным компонентом библиотеки C++, стандартным или нестандартным, и полезны для понимания того, как они работают. Это определенно не самое важное при изучении C++, но эти операции интересны.

Перегрузка операций: краткий обзор

Перегруженная операция представляет собой функцию, переопределенную программистом, со специальным именем, составленным из зарезервированного слова `operator` и символа или символов операции. Кроме того, перегруженные операции известны под именами операторных функций, перегруженных операторных функций или просто операций. Они обеспечивают удобный синтаксис операций для манипулирования объектами классов, определенных программистами.

Включение перегруженных операций в язык C++ было вызвано желанием интерпретировать переменные встроенного типа. Если можно добавить два числовых значения, добавьте два объекта `Account`. Если можно добавить три числовых значения, добавьте три объекта `Account` и т. д. Перегруженные операции позволяют это сделать.

Именно в языке C++ значение встроенных операций определяется по встроенным типам. Для перегруженных операций это делает программист. Значение не должно быть произвольным. Оно должно зависеть от характера добавляемых,

умножаемых объектов. Но программист располагает достаточной свободой в определении смысла операции, и это может легко привести к неправильному использованию — к проектированию перегруженных операций, значение которых не очень понятно интуитивно. Хорошим примером такого неверного использования является унарная операция, к которой добавлена операция, спроектированная в главе 10, для отображения полей комплексных чисел (см. листинг 10.4). Если вы будете ее использовать в клиентской программе, то любой программист, осуществляющий сопровождение, окажется в сложной ситуации. Немногие люди могут правильно угадать, что, например, `+x` означает отображение полей объекта `x` в указанном формате.

Вы не можете свободно выбирать имена перегруженных операций. Имя операторной функции должно включать зарезервированное слово `operator`, за которым следует допустимая операция C++ (разрешается использование двухсимвольных операций типа `==` или `+=`).

Из этого правила есть пять исключений: "`.`", "`.*`", "`::`", "`?:`" и "`sizeof`". Перегруженные операции могут быть определены либо как члены класса (следовательно, используемые как сообщения), либо как глобальные функции верхнего уровня ("друзья" класса, объекты которого применяются как операнды перегруженных операций). Если операция перегружается как член класса, она может иметь любые подходящие аргументы.

Цель сообщения будет использоваться как первый операнд. Если операция перегружается как глобальная функция, она должна содержать хотя бы один аргумент класса. Она не может иметь аргументы только встроенных типов. Это ограничение не применяется к операциям управления памятью (`new`, `delete` и `delete []`).

Операции, перегруженные в базовом классе, наследуются в производных классах. Очевидно, что эти операции не могут осуществлять доступ к членам, определенным в производных классах, поскольку члены производного класса находятся вне области видимости базового класса. Следовательно, к ним невозможно осуществить доступ из методов базового класса. Перегруженные операции присваивания являются аномалией — они не наследуются производными классами. Они могут осуществить доступ только к базовой части производного объекта, но не к его производной части. Значит, каждый класс в иерархии наследования должен определять свой собственный оператор присваивания.

Предшествование операций для перегруженных операций такое же, как и для их встроенных аналогов. Например, операция умножения всегда имеет более высокий приоритет, чем операция сложения, какое бы ни было значение для класса, определенного программистом. Синтаксис выражения для перегруженных операций аналогичен соответствующим встроенным операциям. Например, бинарные операции всегда появляются между их двумя аргументами независимо от того, встроены они или перегружены. (Однако в этой главе можно будет увидеть некоторые исключения из этого правила.)

Арность (количество операндов) для перегруженных операций и для соответствующих встроенных операций одинакова. Бинарные операции остаются бинарными, для них требуются два операнда. Как глобальные функции-члены (например, "друзья") бинарные перегруженные операции должны иметь два параметра. Как функции-члены класса бинарные перегруженные операции должны содержать только один параметр, поскольку другой параметр становится целевым объектом сообщения.

Подобным образом, унарные встроенные операции остаются унарными, когда они перегружаются. Если унарная перегруженная операция реализована как глобальная унарная операция, не являющаяся членом класса (например, "друг"), то она будет содержать один параметр. Если эта перегруженная операция определяется как функция-член (отправляемая как сообщение целевому объекту), она не будет иметь параметров.

В качестве простого примера рассмотрим класс Account. Класс сохраняет информацию об имени владельца и текущем балансе счета, а также поддерживает сервисы, которые позволяют клиентской программе осуществлять доступ к значениям элементов данных объекта, вносить вклады и снимать деньги.

В дополнение к четырем расположенным внутри текста функциям-членам класс имеет общий конструктор. Для класса не требуется конструктор по умолчанию, поскольку объекты Account будут создаваться в динамически распределяемой области памяти, когда они нужны. Конструктор по умолчанию может быть полезен, если объекты класса были созданы заранее, когда имя владельца и исходный баланс еще не были известны.

Поскольку класс динамически управляет памятью, хорошо было бы добавить к нему копию конструктора и оператор присваивания или сделать закрытыми прототипы этих функций-членов (см. главу 11). Здесь это не показано, поскольку объекты Account не передавались по значению. Один объект Account не инициализировался из данных другого объекта Account и один объект Account не присваивался другому объекту Account. В реальной жизни важно защитить объекты Account даже от случайного неверного использования.

```
class Account {                              // базовый класс иерархии
protected:
  double balance;                            // защищенные данные
  char *owner;
public:

  Account(const char* name, double initBalance)      // общий
  { owner = new char[strlen(name)+1];        // выделение памяти для динамически
                                             // распределяемой области памяти
    if (owner == 0) { cout << "\nOut of memory\n"; exit(0); }
    strcpy(owner, name) ;                    // инициализация полей данных
    balance = initBalance; }

  double getBal() const                      // общий для обоих счетов
  { return balance; }

  const char* getOwner() const               // защитить данные от изменений
  { return owner; }

  void withdraw(double amount)
  { balance -= amount; }                     // извлечение обязанностей

  void deposit(double amount)
  { balance += amount; } } ;                 // безусловное приращение
```

Предполагалось создать массив указателей Account, динамически задать объекты Account, инициализировать их, выполнить поиск по счетам, принадлежащим указанному владельцу, внести и снять некоторые денежные суммы. Для упрощения примера снова будут использоваться "жестко заданные" данные, а не данные, загруженные из внешнего файла.

В листинге 16.1 приведена исходная программа для данного примера. Функция createAccount() динамически создает объект Account, вызывает конструктор Account с двумя параметрами и возвращает указатель на новый созданный объект. Функция processRequest() устанавливает флаги ios для вывода на печать чисел с плавающей точкой в фиксированном формате и с нулевыми младшими разрядами числа, выполняет поиск имени клиента в рамках объектов и выводит на печать сообщение, если имя не найдено. В противном случае функция запрашивает у пользователя код транзакции, сумму транзакции и выполняет транзакцию на указанную сумму (вклад или снятие).

Листинг 16.1. Пример обработки класса Account методами, заданными программистами

```
#include <iostream>
using namespace std;

class Account {                              // базовый класс иерархии
protected:
  double balance;                            // защищенные данные
  char *owner;
public:

  Account(const char* name, double initBalance)        // общий
  { owner = new char[strlen(name)+1] ;       // выделить пространство из динамически
                                             // распределяемой области памяти
    if (owner == 0) { cout << "\nOut of memory\n" exit(0); }
    strcpy(owner, name);                     // инициализировать поля данных
    balance = initBalance; }

  double getBal() const                      // общее для обоих счетов
  { return balance; }

  const char* getOwner() const               // защита данных от изменений
  { return owner; }

  void withdraw(double amount)
  { balance -= amount; }                     // извлечение ответственности

  void deposit(double amount)
  { balance += amount; }                     // безусловное приращение
} ;

Account* createAccount(const char* name, double bal)
{ Account* a = new Account (name, bal);      // счет в динамически распределяемой области памяти
  if (a == 0) { cout << "\nOut of memory\n"; exit(0); }
  return a; }

void processRequest(Account* a[] , const char name[])
{ int i; int choice; double amount;
  cout. setf (ios:: fixed, ios: :floatfield);
  cout.precision(2);
  for (i=0; a[i] != 0; i++)
  { if (strcmp(a[i]->getOwner(),name)==0)                          // поиск имени
      { cout << "Account balance: " << a[i]->getBal () << endl;
        cout <<"Enter 1 to deposit, 2 to withdraw, 3 to cancel: ";
        cin >> choice;                                             // тип транзакции
        if (choice != 1 && choice != 2) break;                     // выход
        cout << "Enter amount: ";
        cin >> amount;                                             // сумма транзакции
        switch (choice) {                                          // выбрать следующий путь
            case 1: a[i]->deposit(amount);                         // безусловно
                    break;
            case 2: if (amount <= a[i]->getBal())                  // достаточно средств?
                       a[i]->withdraw(amount);
                    else
                       cout << "Insufficient funds\n" ;             // конец области switch
                    break; }
        cout << "New balance: "<< a[i] ->getBal() << endl;         // OK
        break; } }                                                 // конец цикла поиска
  if (a[i] == 0)
    { cout << "Customer is not found\n"; } }
```

```
int main()
{
  Account* accounts[100]; char name[80];              // данные программы
  accounts[0] = createAccount("Jones",5000);          // создать объекты
  accounts[1] = createAccount("Smith",3000);
  accounts[2] = createAccount("Green",1000);
  accounts[3] = createAccount("Brown",1000);
  accounts[4] = 0;
  while (true)                                        // запросы процесса
  { cout << "Enter customer name ('exit' to exit) : ";
    cin >> name;                                      // принять имя
    if (strcmp(name,"exit")==0) break;                // выход?
    processRequest(accounts, name);                   // следующая транзакция
        }
  return 0;
  }
```

```
Enter customer name ('exit' to exit): Brown
Account balance: 1000.00
Enter 1 to deposit, 2 to withdraw, 3 to cancel:2
Enter amount: 2000
Insufficient funds
New balance: 1000.00
Enter customer name ('exit' to exit): Brown
Account balance: 1000.00
Enter 1 to deposit, 2 to withdraw, 3 to cancel: 2
Enter amount: 500
New balance: 500.00
Enter customer name ('exit' to exit): Smith
Account balance: 3000.00
Enter 1 to deposit, 2 to withdraw, 3 to cancel: 1
Enter amount: 2000
New balance: 5000.00
Enter customer name ('exit' to exit): Simons
Customer is not found
Enter customer name ('exit' to exit): exit
```

Рис. 16.1. *Вывод для программы, представленной в листинге 16.1*

Функция main() определяет массив указателей Account и вызывает createAccount() для создания объектов Account. В цикле она запрашивает у пользователя имя клиента и вызывает функцию processRequest() для обработки транзакции. Пример выполнения программы показан на рис. 16.1.

В данном примере класс Account основывается на своей клиентской программе для проверки, является ли законной транзакция снятия. Преимущество подобного подхода в том, что функции-члены Account не включены в пользовательский интерфейс, они отвечают только за доступ к элементам данных Account. Недостаток этого подхода состоит в том, что данные выталкиваются в клиентскую программу для дальнейшей обработки, а не передают обязанность серверной программе. Такая структура лучше подходит к использованию перегруженных операций.

Первыми кандидатами на роль перегруженных операций являются функции-члены Account — deposit() и withdraw(). Для преобразования их в функции-операторы надо отбросить текущие имена (deposit и withdraw) и переместиться под новые имена (operator+= и operator-=). Другие изменения не требуются.

```
void operator -= (double amount)
{ balance -= amount; }              // клиент проверяет выполнение

void operator += (double amount)
{ balance += amount; }              // безусловное приращение
```

Вместо вызова функций-членов deposit() и withdraw() клиентская функция processRequest() сможет использовать синтаксис выражения, в котором операция вставлена между левым операндом (цель сообщения) и правым операндом (параметр сообщения).

```
switch (choice) {
  case 1: *a[i] += amount;          // a[i]->вклад(сумма);
          break;
```

```
case 2: if (amount <= a[i]-> getBal())
           *a[i] -= amount;              // a[i]->снятие(сумма);
        else
           cout < "Insufficient funds\n";
        break; }
```

Обратите внимание, что цель сообщения — указатель Account. Он должен быть разыменован, когда используется в выражениях. Это неудобство, но не очень серьезное.

Реальный смысл синтаксиса выражения, конечно, вызов функции для отправления сообщения левому операнду в таком выражении: `a[i]->operator+=(amount)` или `a[i]->operator-=(amount)`.

```
Customer List:

Jones                          5000.00
Smith                          3000.00
Green                          1000.00
Brown                          1000.00

Enter customer name ('exit' to exit): Smith
Recount balance: 3000.00
Enter 1 to deposit, 2 to withdraw, 3 to cancel: 1
Enter amount: 1000
New balance: 4000.00
Enter customer name ('exit' to exit): exit
```

Рис. 16.2. *Вывод программы из листинга 16.2*

В листинге 16.2 представлена программа, использующая перегруженные операторные функции вместо именованных программистом методов. Это просто для программы листинга 16.1. Прежде чем начать этап интерактивной обработки, функция `main()` вызывает функцию `printList()`, которая проходит по списку указателей Account и выводит на печать содержание объектов, на которые указывает указатель (см. рис. 16.2). Обратите внимание на то, что операторы форматируют имена, выводимые на печать с выравниванием по левому краю, а остатки на счетах выводятся на печать с выравниванием по правому краю.

Подобно `processRequest()`, функция `printList()` итеративно выполняется по списку до тех пор, пока в массиве не будет найден нулевой указатель (он играет роль сигнальной метки). Обратите внимание, что заголовки циклов этих двух функций различаются. В `printList()` индекс `i` является локальным для цикла, в `processRequest()` — глобальным для цикла. (Он является локальным для области видимости функций.) Причина различий состоит в том, что после завершения цикла в `printList()` значение индекса больше не требуется. Итерации всегда выполняются от начала списка до конца. В `processRequest()` итерацию можно остановить до того, как будет достигнут конец списка (если имя найдено), а `processRequest()` должен знать об этом.

Листинг 16.2. Пример обработки класса Account методом перегрузки операции

```
#include <iostream>
using namespace std;

class Account {                                  // базовый класс иерархии
protected:
  double balance;                                // защищенные данные
  char *owner;
public:

  Account(const char* name, double initBalance)  // общий
  { owner = new char[strlen(name)+1];            // динамическое выделение области памяти
    if (owner ==0) { cout << "\nOut of memory\n" exit(0); }
       strcpy(owner, name);                      // инициализация полей данных
       balance = initBalance; }

  double getBal() const                          // общее для обоих счетов
  { return balance; }
```

```
   const char* getOwner() const                    // защита данных от изменений
   { return owner; }

   void operator -= (double amount)
   { balance -= amount; }                          // извлечение ответственности

   void operator += (double amount)
   { balance += amount; }                          // безусловное приращение
} ;

Account* createAccount(const char* name, double bal)
{  Account* a = new Account(name,bal);             // счет в динамически выделяемой области
   if (a == 0) { cout < "\nOut of memory\n"; exit(0); }
   return a; }

void processRequest(Account* a[], const char name[])
{  int i; int choice; double amount;
   cout.setf(ios::fixed,ios::floatfield);
   cout.precision(2) ;
   for (i=0; a[i] != 0; i++)
   { if (strcmp(a[i]->getOwner(),name)==0)         // поиск имени
        {  cout << "Account balance: " << a[i]->getBal() << endl;
           cout <<"Enter 1 to deposit, 2 to withdraw, 3 to cancel: ";
           cin >> choice;                          // тип транзакции
           if (choice != 1 && choice != 2) break;
              cout << "Enter amount: ";
              cin >> amount;                       // сумма транзакции
              switch (choice) {
                 case 1: *a[i] += amount;          // a[i]->operator+=( сумма);
                         break;
                 case 2: if (amount <= a[i]->getBal())
                            *a[i] -= amount;       // a[i]->operator-=( сумма);
                         else
                            cout >> "Insufficient funds\n";
                         break; }                  // конец области действия switch
           cout << "New balance: "<< a[i]->getBal() << endl;
        break; } }                                 // конец цикла поиска
    if (a[i] == 0)
       { cout << "Customer is not found\n"; } }

void printList (Account* a[])
{  cout << "Customer List:\n\n";
   for (int i=0; a[i] != 0; i++)
   { cout.setf(ios::left, ios::adjustfield); cout.width(30);
     cout << a[i]->getOwner();
     cout.setf(ios::right, ios::adjustfield); cout.width(10);
     cout << a[i]->getBal() << endl; }
   cout << endl; }

int main()
{
   Account* accounts[100]; char name[80];           // данные программы
   accounts[0] = createAccount("Jones",5000);       // создать объекты
   accounts[1] = createAccount("Smith",3000);
   accounts[2] = createAccount("Green",1000);
   accounts[3] = createAccount("Brown",1000);
   accounts[4] = 0;
   printList(accounts);
```

```
while (true)                                    // запросы процесса
{ cout << "Enter customer name ('exit' to exit): ";
  cin >> name;                                  // принять имя
     if (strcmp(name,"exit")==0) break;         // выход?
     processRequest(accounts, name);            // следующая транзакция
     }
return 0 ;
}
```

Реализовать перегруженные операции как глобальные функции просто. Цель сообщения становится первым параметром функции. Вместо элементов данных целевого объекта операции используют элементы данных первого параметра. Ниже приведены две операции, реализованные как глобальные функции.

```
void operator -= (Account &a, double amount)     // глобальная функция
  { a.balance -= amount; }                       // извлечение ответственности

void operator += (Account &a, double amount)
  { a.balance += amount; }                       // безусловное приращение
```

Поскольку эти две функции осуществляют доступ к неоткрытым компонентам класса Account, они должны быть объявлены "друзьями" класса Account. У некоторых программистов это вызывает раздражение, поскольку требует дополнительной работы. Как уже упоминалось, современный подход к программированию не рассматривает дополнительные затраты на написание программы как недостаток. Дополнительные данные записываются только один раз, но читаются в ходе разработки, тестирования и сопровождения программы неоднократно.

В этом случае добавление описаний функций, "дружественных" для класса, четко показывает, что эти функции принадлежат заданному классу. Учтите, что они не могут использоваться без объектов класса Account. Функции-"друзья" принадлежат классу концептуально, т. е. являются частью операций, предоставляемых классом. Синтаксис функций-"друзей" отличается от синтаксиса функций-членов. Частые обвинения против использования "дружественных" функций, нарушения инкапсуляции и создание дополнительных зависимостей между частями программы не являются результатом применения перегруженных операторных функций.

```
class Account {                                  // базовый класс иерархии
protected:
       double balance;                           // защищенные данные
       char *owner;
public:
  Account(const char* name, double initBalance)     // общие
  { owner = new char[strlen(name)+1];            // динамическое выделение
                                                 // распределяемой области
    if (owner == 0) { cout << "\nOut of memory\n"; exit(0); }
    strcpy(owner, name);                         // инициализация полей данных
    balance = initBalance; }
  double getBal() const                          // общие для обоих счетов
  { return balance; }
  const char* getOwner() const                   // защита данных от изменений
  { return owner; }
friend void operator-= (Account &a, double amount);    // операторы
friend void operator+= (Account &a, double amount);
} ;
```

Синтаксис выражения в клиентской программе не изменяется с переходом от операций функций-членов к операциям-"друзьям".

```
switch (choice) {
  case 1: *a[i] += amount;                    // operator+=(*a[i],amount);
          break;
  case 2: if (amount <= a[i]->getBal())
             *a[i] -= amount;                 // operator-=(*a[i],amount);
          else
             cout << "Insufficient funds\n";
          break; }                            // конец области действия switch
```

Смысл этой программы изменяется. Синтаксис выражения — все еще синтаксическое смягчение вызова функции, но это вызов глобальной функции. В вызове функции отсутствует необходимость в целевом объекте. Вместо этого объект, который участвует в операции, передается как фактический аргумент функции. Так компилятор воспринимает эту клиентскую программу.

```
switch (choice) {
  case 1: operator+=(*a[i],amount);           // a.k.a. *a[i]+=сумма;
          break;
  case 2: if (amount <= a[i]->getBal())
             operator-=(*a[i],amount);        // a.k.a. *a[i]-=сумма;
          else
             cout << "Insufficient funds\n";
          break; }                            // конец области действия switch
```

Обратите внимание, что фактический параметр должен быть разыменован, поскольку a[i] является указателем на объект Account, но не самим объектом. Параметр-ссылка должен инициализироваться значением объекта, а не значением указателя. Именно поэтому вызов функции должен быть разыменован.

Использование перегруженных операций предлагает очень хороший способ записи клиентской программы. Однако таким образом существенные вопросы проектирования ПО не решаются. Все, что можно выполнить с использованием перегруженных операций, можно сделать и с помощью обычных функций-членов (см. листинг 16.1).

Унарные операции

Унарные операции содержат только один операнд. К ним относятся операции инкремента и декремента, операции отрицания, логического и побитового отрицания, операции плюс и минус, приведения, адресации, операции разыменования и операторы new, delete и sizeof. Все они (за исключением sizeof) могут быть перегружены.

Не все операции могут иметь свое специальное значение для каждого класса. В главе 10 операция плюса для класса Complex перегружена как операция вывода, и эта структура сбивает с толку. Именно поэтому перегрузка унарной операции не очень популярна. Однако бывают ситуации, когда эти операции могут помочь в понимании клиентской программы. Далее обсудим несколько примеров перегруженных унарных операций.

Операции инкремента и декремента

Операции инкремента и декремента очень популярны в C++. Они используются в обработке текстов, когда приращение (или уменьшение) указателя может объединяться с доступом к текущему символу для обработки.

```
void printString(const char data[])     // текст не изменяется
{ const char *p = data;                 // указывает на начало данных
  while (*p != 0)                       // выполняется до конца данных
  { cout << *p;                         // вывод на печать текущего символа
    ++p; }                              // вывод на печать следующего символа
  cout << endl; }
```

В данном примере массив символов передается глобальной функции как константа, и каждый символ отображается по очереди. Указатель p устанавливается вначале на первый символ массива data[], а затем увеличивается, пока не укажет на оконечный ноль. Даже если это выглядит как логическая структура очень низкого уровня, увеличивающая адрес памяти, в действительности такая операция скорее абстрактная, потому что она не показывает реальные детали управления памятью — насколько изменяется адрес и увеличивается или уменьшается он фактически.

Например, отсутствует гарантия того, что массив параметров размещается в памяти от нижних адресов к верхним. Возможно, как и у автора, физические адреса к концу массива уменьшаются. Следовательно, отсутствует гарантия, что содержимое указателя фактически увеличивается, когда к указателю применяется операция инкремента. В данном случае подразумевается, что указатель устанавливается на доступ к следующему компоненту массива независимо от размера компонента.

Тем не менее этот "открытый" доступ к массиву компонентов, предоставленный клиентской программой, подвержен ошибкам. Доступ к ячейкам, расположенным за пределами границ массива, не помечается во время компиляции как синтаксические ошибки. Выполнение такого доступа может вызвать аварийное завершение программы на этапе выполнения. Она может скрыто выдавать верные результаты до некоторого момента, когда изменится использование памяти и произойдет катастрофа.

Объединение данных и операций в одном классе защищает данные от выполнения нелепого доступа клиентскими программистами и предоставляет программистам инструмент для обработки объектов, который защищает от возникновения ошибок. Ниже приведен пример класса String, подобный уже рассмотренному в главе 11.

```
class String {
  int size;                             // размер строки
  char *str;                            // начало внутренней строки
  void set(const char* s) ;             // выделение закрытой строки
public:
  String (const char* s = "")           // по умолчанию и преобразование
    { set(s); }
  String (const String &s)               // конструктор копирования
    { set(s.str); }
  ~String()                              // деструктор
  { delete [ ] str; }
  String& operator = (const String& s);  // присваивание
int getSize() const;                     // длина текущей строки
char* reset() const; } ;                 // установка на начало строки
```

Элемент данных класса str указывает на динамически выделенный массив, размер которого сохраняется в элементе данных size. Закрытая функция-член set() используется конструкторами класса и оператором присваивания. Она принимает массив функций как параметр и динамически выделяет память из динамически распределяемой области памяти. Затем она устанавливает указатели

элемента данных str на эту заново выделенную память и динамически инициализирует выделенную память, используя текстовый массив, предоставленный как аргумент.

```
void String::set(const char* s)
  { size = strlen(s);                  // оценка размера
    str = new char[size + 1];          // запрос памяти динамически
                                       // распределяемой области
    if (str == 0) { cout << "Out of memory\n"; exit(0); }
    strcpy(str,s); }                   // копирование клиентских данных в "кучу"
```

Это типичная структура динамического управления памятью. Такая закрытая функция удобна, потому что она инкапсулирует операции, общие для конструкторов и оператора присваивания.

Как подобает классу, который динамически управляет своей памятью, класс String предоставляет конструктор преобразования (он дублируется как конструктор по умолчанию), конструктор копирования, оператор присваивания и деструктор. Конструктор по умолчанию передает set() пустую строку. Конструктор преобразования отправляет set() как параметр в свой массив символов. Конструктор копирования передает set() в динамически распределяемую область памяти своего объекта-параметра. Деструктор возвращает память динамически распределяемой области памяти, которую конструктор или оператор присваивания выделил для объекта String.

Такое добавление функций-членов поддерживает использование объектов String в разнообразных контекстах. Клиентская программа может определить объект String как неинициализированную переменную (вызывается конструктор по умолчанию), как объект, инициализированный значением символьного массива (применяется конструктор преобразования), или как объект, инициализированный данными другого существующего объекта String (вызывается конструктор копирования).

Оператор присваивания поддерживает присваивание объекта для самого себя. Он освобождает существующую память динамически распределяемой области памяти и выделяет и инициализирует новую память динамически распределяемой области памяти с помощью set(). Он поддерживает клиентскую программу, которая использует синтаксис выражения для нескольких последовательных присваиваний (возвращая ссылку на целевой объект String). Обратите внимание, что хотя тело присваивания возвращает полный объект String (в данном случае разыменованный указатель), это только ссылка на возвращаемый объект — копирование отсутствует.

```
String& String::operator = (const String& s)
{ if (this == &s) return *this;      // ничего, если поддерживается
                                     // самоприсваивание
  delete [ ] str;                    // возвращение существующей памяти
  set(s.str);                        // выделение/установка новой памяти
  return *this; }                    // для поддержки цепочечного присваивания
```

В листинге 16.3 показана полная реализация класса String вместе с встроенными функциями-членами getSize() и reset(). Первая функция возвращает максимальное количество символов, которые может содержать объект String. Вторая функция возвращает указатель на внутреннюю строку так, что клиентская программа (функции printString() и modifyString()) может инициализировать внешний указатель, который ссылается на внутреннюю строку.

Эти две клиентские функции используют операцию инкремента для поиска и замены символов внутри их параметра объектов String. Цикл в printString() продолжается до тех пор, пока во внутренней строке параметра String не встретится оконечный нуль. (Указатель p на символ внутренней строки должен быть

```
Hello World!
How is the weather?
```

Рис. 16.3. *Пример нарушения целостности информации в памяти программой из листинга 16.3*

разыменован.) Цикл в modifyString() работает до тех пор, пока все символы в параметре массива символов text[] не будут скопированы.

Функция main() создает и инициализирует объект String, выводит на печать его содержимое, затем изменяет его и осуществляет вывод на печать. Поскольку цикл в modifyString() не принимает во внимание текущий объем памяти динамически распределяемой области памяти, выделенный для параметра String, это приводит к нарушению целостности информации в памяти. Вывод программы представлен на рис. 16.3.

Листинг 16.3. Пример использования операции инкремента с указателем на внутренние данные

```
#include <iostream>
using namespace std;

class String {
  int size;                              // размер строки
  char *str;                             // начало внутренней строки
  void set(const char* s);               // выделение закрытой строки
public:
  String (const char* s = "")            // по умолчанию и преобразование
    { set(s); }
  String (const String &s)               // конструктор копирования
    { set(s.str); }
  ~String()                              // деструктор
  { delete [ ] str; }
  String& operator = (const String& s);  // присваивание
  int getSize() const;                   // длина текущей строки
  char* reset() const; };                // сброс в начало строки

void String::set(const char* s)
  { size = strlen(s);                    // оценка размера
    str = new char[size + 1];            // запрос динамически распределяемой области памяти
    if (str == 0) { cout << "Out of memory\n"; exit(0); }
       strcpy(str,s); }                  // копирование клиентских данных
                                         // в динамически распределяемую область памяти

String& String::operator = (const String& s)
{ if (this == &s) return *this;          // ничего, если поддерживается самоприсваивание
  delete [ ] str;                        // возвращение существующей памяти
  set(s.str);                            // выделение/установка новой памяти
  return *this; }                        // для поддержки цепочечного присваивания

int String::getSize() const              // нет изменений объекта String
{ return size; }

char* String::reset() const              // нет изменений объекта String
{ return str; }                          // возвращение указателя к началу

void printString(const String&: data)    // нет изменений в строке
{ char *p = data.reset();                // указатель на первый символ
  while (*p != 0)                        // выполнение до обнаружения конца символов
  { cout << *p;                          // вывод на печать текущего символа
    ++p; }                               // указатель на следующий символ
  cout << endl; }

void modifyString(const String& data, const char text[])       // плохой
{ char *p = data.reset();                // указатель на первый символ
  int len = strlen(text) + 1;            // установка предела итерации
```

```
  for (int i=0; i < len; i++)          // просмотр каждого символа
  { *p = text[i];                      // копирование текущего символа
    ++p; } }                           // указатель на следующий символ

int main()
{
  String data = "Hello World!";
  printString(data);                   // хороший вывод
  modifyString(data,"How is the weather?");
  printString(data);                   // память искажается
  return 0;
  }
```

Эту проблему совсем несложно исправить. Клиентская программа `modifyString()` должна проверять доступное пространство и прекращать перекачивание данных в объект при достижении границ.

```
void modifyString(const String&: data, const char text[])      //ok
{ char *p = data.reset();              // указатель на первый символ
  int len = strlen(text) + 1;          // установка предела итерации
  int size = data.getSize();           // установка другого предела итерации
  for (int i=0; i<len && i<size; i++)  // просмотр каждого символа
  { *p = text[i];                      // копирование текущего символа
    ++p; } }                           // указатель на следующий символ
```

Эта функция `modifyString()` исключает проблему нарушения целостности информации в памяти, однако сохраняются дефекты проектирования. Клиентская программа должна понимать детали структуры сервера (в данном случае `String`). Решение этой проблемы с использованием функций перегруженных операций может быть полезным.

Структура `String` обязана отражать изменение отношения. Чтобы быть в состоянии защитить объект от неверного использования клиентами, класс `String` должен сохранить состояние своих объектов. В этом случае состояние должно включать указатель на текущий символ, выводимый на печать или изменяемый.

Рассматривался важный метод проектирования в C++. Когда будет принято решение, какие данные должен сохранять класс, включайте элементы данных, которые отражают состояние объекта для клиента. Данный метод освободит объекты от этой зависимости (что не было сделано в листинге 16.3). Покажем лучший вариант класса `String`.

```
class String {
  int size;                            // размер строки
  char *str;                           // начало внутренней строки
  char *ptr;                           // указатель на текущий символ
  void set(const char* s);             // выделение закрытой строки
public:
  String (const char* s = "")          // по умолчанию и преобразование
    { set(s); }
  String (const String &s)             // конструктор копирования
    { set(s.str); }
  -String()                            // деструктор
  { delete [ ] str; }
String& operator = (const String&; s); // присваивание
char* operator++();                    // префиксная операция инкремента
int getSize() const;                   // длина текущей строки
char* reset(); } ;                     // не константа: объект изменяется
```

Здесь указатель ptr ссылается на текущий символ. Указатель изменяется по памяти динамически распределяемой области в операции инкремента, которая возвращает адрес текущего символа клиенту для доступа или изменения. Операция инкремента проверяет, смещает ли запрос клиента указатель ptr за пределы динамически распределяемой области памяти символьного массива. Если смещает, то операция не увеличивает указатель. Вместо этого она изменяет символ, на который указывает указатель, в "\0", чтобы обеспечить правильное завершение символьного массива. Снова используется выражение ptr-str<size, но это не означает, что значение адреса в элементе данных ptr действительно больше, чем значение адреса в элементе набора данных str. Это хороший способ показать перемещение указателя.

```
char* String::operator ++()          // приращение, затем доступ
{ if (ptr-str < size)                // проверка, достаточно ли памяти
    return ++ptr;                    // указатель на следующий символ
  else
    {  *ptr = 0;                     // установить оконечный нуль
       return ptr; } }               // не пересылать его, если конец
```

Хорошим местом для инициализации этого указателя является функция-член reset(). Она может вызываться из клиентской программы до начала следующей итерации по тексту. Важное различие между этой и предыдущей структурой в листинге 16.3 состоит в том, что в данном варианте функцию reset() нельзя пометить как константу — изменится состояние объекта. Всегда уделяйте внимание режиму поведения метода и не забывайте соответственно пометить функцию-член. (Это не касается функций, не являющихся членами класса.)

```
char* String::reset()          // не константа: объект изменяется
{ ptr = str;                   // установить текущий указатель на начало
  return str; }                // возвращение указателя на начало
```

Некоторые программисты ощущают неудобство, когда элемент данных не инициализирован в конкретное значение при создании объекта. Такие программисты также инициализировали бы элемент данных ptr при каждом вызове конструктора. В структуре, где каждый конструктор вызывает закрытую функцию set(), инициализация может быть выполнена в данной функции.

```
void String::set(const char* s)
  {  size = strlen(s);              // оценка размера
     str = new char[size + 1];      // запрос динамически распределяемой
                                    // области памяти
     if (str == 0) { cout << "Out of memory\n"; exit(0); }
     strcpy(str,s);                 // копирование клиентских данных в "кучу"
     ptr = str; }                   // инициализация рабочего указателя
```

Что достигается при такой структуре? С момента, когда рождается объект, он готов к итерации. Не требуется явно отправлять ему сообщение reset(). Обязанности перемещаются от клиентской программы, которая должна в предыдущей структуре вызывать reset(), к конструктору объекта.

Эта идея хорошо работает при осуществлении доступа только по чтению к данным объекта String. Функция printString() останавливает выполнение итераций, когда обнаруживается ограничивающий нуль. Операция инкремента может распознать это и установить текущий указатель обратно в начало строки.

```
char* String::operator ++()          // приращение, затем доступ
{ if (ptr-str < size)                // проверка, достаточно ли памяти
    return ++ptr;                    // указатель на следующий символ
```

```
else
   { *ptr = 0;                  // установить оконечный нуль
     ptr = str;                 // снова указатель на начало данных
     return ptr; } }            // не пересылать его, если конец
```

В данной ситуации наличие функции-члена reset() не обязательно. После выполнения каждого сканирования по данным объекта String указатель должен сбрасываться, а объект будет готов к следующему сканированию.

```
void printString(String& data)     // не константа: строка изменяется
{ char *p = data.reset();   // действительно ли нужен этот вызов?
  while (*p != 0)           // выполнение, пока не обнаружится конец символов
  { cout << *p;             // вывод текущего символа
    p = ++data; }           // замечательный синтаксис: объект изменяется
  cout << endl; }
```

Однако в этой структуре функция reset() все еще необходима, так как клиентская функция modifyString() может попытаться осуществить доступ к строковым данным за пределами обозначенных границ. Поскольку операция инкремента не знает, как долго клиентская программа будет предоставлять новые данные для копирования в строку, она не может переместить указатель в начало данных. Все проблемы возникают из-за "высокомерного" характера функции modifyString(), которая закачивает данные в параметр объекта String независимо от наличия в памяти свободного пространства.

```
void modifyString(String& data, const char text[])     // не константа
{ char *p = data.reset();                    // указывает на первый символ
  int len = strlen(text) +1;                 // устанавливает предел итераций
  for (int i=0; i < len; i++)                // просматривает каждый символ
  { *p = text[i];                            // копирует текущий символ
    p = ++data; } }                          // указатель на следующий символ
```

Обратите внимание на то, что данные объекта параметра не содержат модификатор const, но не потому, что функция modifyString() записывает новое содержание в память своей динамически распределяемой области памяти. В листинге 16.3 функция modifyString() выполняет то же самое, но данные параметра помечаются как const. Язык C++ не обращает внимания на изменения в памяти динамически распределяемой области памяти объекта, но он учитывает изменения элементов данных объекта. Операция инкремента (и функция reset()) вызывает изменения в элементе данных ptr и, следовательно, предотвращает использование модификатора const с параметром String.

```
Hello World!
How is the w
```

Рис. 16.4. *Вывод программы из листинга 16.4*

В листинге 16.4 показана полная программа, которая реализует операцию инкремента для класса String. Вывод программы представлен на рис. 16.4. Проблема искажения информации в памяти исчезает.

Листинг 16.4. Пример использования операции инкремента как сообщения объекту.

```
#include <iostream>
using namespace std;

class String {
  int size;                         // размер строки
  char *str;                        // начало внутренней строки
  char *ptr;                        // указатель на текущий символ
  void set(const char* s);          // выделение закрытой строки
 blic:
   ring (const char* s = "")        // по умолчанию и преобразование
    { set(s); }
```

```
   String (const String &s)                    // конструктор копирования
     { set(s.str); }
   ~String()                                   // деструктор
   { delete [ ] str; }
   String& operator = (const String& s);       // присвоение
   char* operator++();                         // префиксная операция инкремента
   int getSize() const;                        // длина текущей строки
   char* reset(); } ;                          // не константа: объект изменяется

void String::set(const char* s)
   { size = strlen(s);                         // оценка размера
     str = new char[size + 1];                 // запрос памяти динамически распределяемой области
     if (str ==0) { cout << "Out of memory\n"; exit(0); }
     strcpy(str,s);                            // копирование клиентских данных в "кучу"
     ptr = str; }                              // инициализация рабочего указателя

String& String::operator = (const String& s)
{  if (this == &s) return *this;               // ничего, если обеспечивается самоприсваивание
   delete [ ] str;                             // возвращение существующей памяти
   set(s.str);                                 // выделение/установка новой памяти
   return *this; }                             // для поддержки цепочечного присваивания

int String::getSize() const                    // объект String не изменяется
{ return size; }

char* String::reset ()                         // не константа: объект изменяется
{  ptr = str;                                  // установить текущий указатель в начало
   return str; }                               // возвращение указателя в начало

char* String::operator ++()                    // приращение, затем доступ
{  if (ptr-str < size)                         // проверка, достаточно ли памяти
       return ++ptr;                           // указатель на следующий символ
   else
       { *ptr = 0;                             // установить оконечный нуль
         return ptr; } }                       // не пересылать его, если конец

void printString(String& data)                 // не константа: строка изменяется
{  char *p = data.reset();                     // указатель на первый символ
   while (*p != 0)                             // переход, пока не обнаружится конец символов
   { cout << *p;                               // копировать текущий символ
     p = ++data; }                             // указатель на следующий символ
   cout << endl; }

void modifyString(String& data, const char text[])
{  char *p = data.reset();                     // указатель на первый символ
   int len = strlen(text) + 1;                 // установить предел итерации
   for (int i=0; i < len; i++)                 // просмотр каждого символа
   { *p = text[i];                             // копировать текущий символ
     p = ++data; } }                           // указатель на следующий символ

int main()
{
   String data = "Hello World!";
   printString(data);                          // хороший вывод
   modifyString(data,"How is the weather?");
   printString(data);                          // память НЕ искажается
   return 0;
   }
```

Использование операции инкремента обеспечивает прекрасный синтаксис. Далее приводятся дополнительные комментарии по поводу примера. Во-первых, перегруженная операция инкремента проверяет граничные условия для индекса, что не может выполнить встроенная операция. При выполнении этого в данной операции обязанности переносятся в серверный класс. Однако нужным этот пример делает именно проверка границ, а не операция инкремента, которая обеспечивает прекрасный синтаксис в `printString()` и `modifyString()`. Кроме того, такая проверка границ может выполняться, если имя функции было `movePointer()` или `next()`, а не `operator++()`.

Во-вторых, эта реализация операции инкремента возвращает указатель на текущий символ в памяти динамически распределяемой области и позволяет клиенту делать с ним все, что ему потребуется. Это опасная и порождающая ошибки практика.

Операции декремента подобны операциям инкремента. В их реализацию не вовлекаются новые принципы или идеи.

Постфиксные перегруженные операции

В C++ встроенные префиксные и постфиксные операции различаются своими позициями относительно операнда выражения. Если выражение записывается как `++data` (данные), это префиксная операция. Если выражение записывается как `data++`, это постфиксная операции. Важно различать их.

Вышесказанное справедливо для перегруженных операций инкремента и декремента. Операция инкремента была реализована как префиксная операция в листинге 16.4. Вначале было изменено состояние целевого объекта, а затем с помощью клиентской программы возвращено новое значение (текущий указатель).

Это не подходит для постфиксных операций. Постфиксная операция для класса `String`, например, должна вначале возвратить текущее значение указателя, а затем увеличить его для будущего использования. Следовательно, должна быть отдельная функция, отличающаяся от операции инкремента, реализованной в листинге16.4.

В соответствии с правилами C++ имя такой отдельной функции должно составляться из зарезервированного слова `operator` и символа, который входит в состав операции (в данном случае либо `++` или `--`). Предположительно имя постфиксного перегруженной операции инкремента должно быть `operator++()`, а имя постфиксной перегруженной операции декремента — `operator--()`.

Но это имена, которые должны были бы использоваться для префиксных перегруженных операций! Наличие двух функций с одинаковыми именами в пределах одной области видимости не допускается, если только они не имеют разные сигнатуры — разное количество параметров неодинаковых типов.

Как упоминалось ранее, количество параметров для перегруженных операций не может быть выбрано произвольно. Если бинарная операция реализована как функция-член, то ее первый операнд является целью сообщения, а второй операнд — параметром сообщения. Если унарная операция реализована как функция-член, то ее единственный операнд должен использоваться как цель сообщения. Такая перегруженная операция не может иметь параметров.

Итак, нам требуется реализовать в одном и том же классе две перегруженные операции с одним именем (например, `operator++`) и одинаковыми сигнатурами (без параметров). В этом случае компилятор может пожаловаться, что их невозможно различить.

Однако программисты C++ требуют, чтобы операции инкремента и декремента можно было реализовывать как в префиксной, так и в постфиксной форме. Для решения этой проблемы C++ предлагает фиктивный целый параметр.

```
char* String::operator ++(int)      // сначала доступ, потом приращение
{ if (ptr-str < size)               // проверка наличия места
    return ptr++;                   // указатель на следующий символ
```

```
else
  {  *ptr = 0;                    // установка оконечного нуля
     return ptr; } }              // если конец, завершить выполнение
```

Роль фиктивного параметра весьма ограниченная. Он должен сообщить компилятору, что эта функция совершенно другая, а не перегрузка операции инкремента (или декремента) без параметров. С другой стороны, этот параметр не имеет значения в теле самой функции. Именно поэтому можно опустить имя параметра — компилятор не покажет, что имя не определено.

В присутствии двух функций компилятор найдет ++data в клиентской программе, интерпретирует это как data.operator++() и выполнит префиксную операцию. Когда в клиентской программе компилятор находит ++data, он интерпретирует ее как data.operator++(0) и выполняет постфиксную операцию. Префиксное значение operator++() и постфиксное значение operator++ (int) не навязываются языком. За них отвечает проектировщик класса.

В листинге 16.5 показана программа из листинга 16.4 с постфиксной перегруженной операцией, которая вызывается из modifyString(). Постфиксная операция возвращает указатель на текущий символ в памяти динамически распределяемой области. Чтобы изменить этот символ, клиентская программа должна разыменовать значение, возвращенное функцией. Получится четкий синтаксис присваивания, который имеет такой же вид, как если бы переменнаяdata была указателем.

```
*data++ = text[i];              // копирование символа
```

Вывод из этой версии программы такой же, как и для листинга 16.4 (см. рис. 16.4).

Листинг 16.5. Пример использования префиксных и постфиксных операций инкремента

```
#include <iostream>
using namespace std;

class String {
  int size;                                 // размер строки
  char *str;                                // начало внутренней строки
  char *ptr;                                // указатель на текущий символ

  void set(const char* s);                  // выделение закрытой строки
public:
  String (const char* s = "")               // по умолчанию и преобразование
    { set(s); }
  String (const String &s)                  // конструктор копирования
    { set(s.str); }
  ~String()                                 // деструктор
  { delete [ ] str; }
  String& operator = (const String& s);     // присваивание
  char* operator++();                       // префиксная операция инкремента
  char* operator++(int);                    // постфиксная операция инкремента
  int getSize() const;                      // длина текущей строки
  char* reset(); } ;                        // не константа: объект изменяется

void String::set(const char* s)
  { size = strlen(s);                       // оценка размера
    str = new char[size + 1];               // запрос памяти динамически распределяемой области
    if (str ==0) { cout << "Out of memory\n" ; exit(0); }
    strcpy(str,s);                          // копирование данных в "кучу"
    ptr = str; }                            // инициализация рабочего указателя
```

```
String& String::operator = (const String& s)
{  if (this == &s) return *this;          // ничего, если самоприсваивание
   delete [ ] str;                        // возвращение существующей памяти
   set (s.str);                           // выделить/установить новую память
   return *this; }                        // для поддержки цепочечных присваиваний

int String::getSize() const               // нет изменений в объекте String
{ return size; }

char* String::reset()                     // не константа: объект изменяется
{  ptr = str;                             // установить текущий указатель на начало
   return str; }                          // возвратить указатель на начало

char* String::operator ++()               // приращение, затем доступ
{  if (ptr-str < size)                    // проверка наличия памяти
      return ++ptr;                       // указатель на следующий символ
   else
      { *ptr = 0;                         // установить оконечный нуль
        return ptr; } }                   // не передавать, если конец

char* String::operator ++(int)            // доступ, затем приращение
{  if (ptr-str < size)                    // проверка наличия памяти
      return ptr++;                       // указатель на следующий символ
   else
      { *ptr = 0;                         // установить оконечный нуль
        return ptr; } }                   // не передавать, если конец

void printString(String& data)            // не константа: строка изменяется
{  char *p = data.reset();                // указатель на первый символ
   while (*p != 0)                        // переход, пока не обнаружится конец символов
   { cout << *p;                          // вывод на печать текущего символа
     p = ++data; }                        // указатель на следующий символ
   cout << endl; }

void modifyString(String& data, const char text[])
{  data.reset();                          // указатель на первый символ
   int len = strlen(text) + 1;            // установить предел итерации
   for (int i=0; i < len; i++)            // просмотр каждого символа
       *data++ = text[i]; }               // замечательный синтаксис: копирование символа

int main()
{
   String data = "Hello World!";
   printString(data);                     // хороший вывод
   modifyString(data,"How is the weather?");
   printString(data);                     // память НЕ искажается
   return 0;
   }
```

Как и в случае префиксных операций инкремента и декремента, перегруженные постфиксные операции улучшают внешний вид программы, но не очень важны с точки зрения программной инженерии.

Операции преобразования

Приведение значения одного типа к значению другого типа достигается за счет применения имени целевого типа для значения (для имени переменной) исходного типа. Синтаксис бывает двух видов: традиционный и новый функционально-подобный. В традиционном синтаксисе имя типа цели (в скобках) используется перед

исходным значением (или переменной). В новом функционально-подобном синтаксисе имя целевого типа применяется, как если бы это была функция с одним параметром, а исходное значение (или переменная) используется как фактический аргумент функции.

```
int x; double y;
x = int ('A');          // функционально-подобный синтаксис; x содержит 65
y = (double)x;          // традиционный синтаксис, y содержит 65.0
double *p = &y;         // правильный тип указателя: это безопасно
int *q = (int*)p;       // int q указывает на double y: тревога
```

Единственное различие между этими двумя формами приведения типа состоит в том, что традиционный синтаксис может использоваться с любым допустимым именем типа, а функционально-подобный синтаксис требует идентификатор для имени типа. Поэтому в последнем примере и представлено два примера приведения числовых типов и только один пример приведения типов указателей. Имя `int*` является допустимым именем типа, но не допустимым идентификатором.

C++ позволяет приведение любых числовых типов без каких-либо ограничений. Приведения могут быть явными (с использованием операции приведения) или неявными (без применения операции).

```
int x; double y;
x = 'A'; y = x;         // явное приведение: в C++ без проблем
```

C++ также допускает приведение произвольных указателей (и ссылок), включая любые типы, определенные пользователем. Эти преобразования могут выполняться только при использовании явного приведения. Неявное приведение указателей не допускается. Используя такие приведения, вы создадите себе проблемы. В данной части программы указатель `String` ссылается на целое значение. Указатель `String` может законно отвечать на любое сообщение `String`. Выполняя это, он интерпретирует память, как если бы она принадлежала объекту `String`. Поскольку первым элементом данных `String` является целое `size`, при запуске объекта `String` сообщение `getSize()` извлечет это значение. Фактически значение находится в переменной `z` целого типа. Если первый элемент данных в классе `String` не был бы целого типа, то программа отобразила бы "мусор".

```
int z = 42; String *ptr = (String*) &z;          // создаем проблемы
cout <<"Size: " <<ptr->getSize() <<endl;         // выводит 42
```

В этой части программы указатель целого типа ссылается на объект `String`. Указатель интерпретирует память объекта `String`, как если бы это было целое. Значение, извлеченное указателем, может использоваться в любом выражении, которое требует целое значение. В данном примере объект `String` при запуске содержит элемент данных `size` целого типа и такое значение извлекается целым указателем. Если бы класс `String` начинался не с элемента данных целого типа, то данная часть программы выводила бы на печать чепуху.

```
int *r; String s("Hello, World!");
r = (int*) &s;
cout << "String: " << 2 + *r << endl;            // выводит 15
```

```
Hello World!
How is the w
Size: 42
String: 15
```

Рис. 16.5. *Вывод для программы из листинга 16.5 с добавленными двумя частями программы*

На рис. 16.5 показан вывод программы из листинга 16.5, в конец которой добавлены эти две части. Эти операции интерпретируют распределение памяти структуры `String` правильно и извлекают первое значение объекта `String`, которое является элементом данных целого типа. Это допускается в C++, но с точки зрения программной инженерии представляет собой кошмар для программиста, осуществляющего сопровождение. Если изменить порядок элементов данных в определении класса, не внося никаких изменений в программу, вывод будет совсем другим.

Разрешение выполнять приведение произвольных указателей и ссылок не распространяется на объекты. Преобразование целых в объекты класса, определенного программистом, не допускается. Невозможно выполнять преобразование объекта класса, заданного программистом, в целое, или в другое числовое значение, или в объект другого класса.

```
String s;  Account a;  int x;
x = s;  a = x;  s = a;                    // это бессмыслица
```

C++ допускает приведение указателей или ссылок классов, связанных наследованием. Приведения такого типа могут быть неявными, если цель приведения (указатель или ссылка) является общедоступным базовым классом и источником приведения (значением, указателем или ссылкой), а также если класс открыто порождается из цели приведения. Оно особенно полезно, когда указание объектов различных производных классов осуществляется массивом указателей базового класса. Вставка этих объектов в массив может выполняться без использования явного преобразования.

Приведение из базового указателя (или ссылки) к указателю производного класса (или ссылке) должно быть явным, подобно приведению несвязанных типов. Это особенно полезно, когда указание объекта производного класса осуществляется указателем (или ссылкой) базового класса, но должно выполнять операции, определенные в производном классе, а не в базовом классе. Явное приведение указывает, к какому производному классу принадлежит запрашиваемая операция.

Приведение базового объекта к производному объекту не допускается. Это подобно интерпретации объектов несвязанных типов. Если необходимо, такое преобразование может предоставляться добавлением конструктора преобразования к производному классу. Этот конструктор содержит параметр базового класса. Примеры таких приведений и конструкторов были описаны в главе 15.

Для классов, связанных наследованием, C++ допускает ослабление строгого контроля типов. Несмотря на то, что не допускается неявное приведение базового объекта к производному объекту, разрешаются неявные преобразования объектов производного класса в объекты базового класса. Если не используется явное приведение, то дополнительные элементы данных (и операции) производных объектов отбрасываются.

C++ поддерживает строгий контроль за типами только для объектов классов, созданных программистом. Для остальных типов преобразование допускается. Некоторые преобразования могут выполняться только явно, в операции приведения (для указателей и ссылок любых типов). Другие преобразования могут осуществляться даже неявно, без явного использования приведения (между числовыми значениями или из объектов, указателей и ссылок производных типов в объекты, указатели и ссылки базового типа). Эти преобразования предоставляют программисту дополнительные возможности при реализации алгоритмов, которые обрабатывают значения разных типов. Поскольку из-за данных преобразований нарушается строгий контроль типов, они устраняют защиту, предоставляемую проверкой синтаксиса.

Однако этого недостаточно. C++ позволяет программистам реализовать дополнительное приведение для объектов классов той категории, для которой поддерживается строгий контроль типов. Обратите внимание, что защита вследствие выполнения проверки синтаксиса обсуждается только для указанных классов. Программист с помощью конструкторов и операции преобразования показывает классы, для которых устанавливается защита.

Конструкторы преобразований описываются в главе 9. Конструктор преобразования содержит один параметр типа, который должен преобразовываться в указанный класс. Например, класс `String` содержит конструктор преобразования, который переводит значение символьного массива в значение `String`.

```
String (const char* s)              // конструктор преобразования
  { set(s); }
```

При наличии конструктора массив символов может использоваться, когда ожидается объект String без возникновения синтаксической ошибки на этапе компиляции. Поскольку String является классом, определенным программистом, то ситуации, в которых появляется объект String, редки. Они ограничиваются определением объекта, передачей параметров по значению (не по указателю или по ссылке), присваиванием и отправкой сообщений объекту.

```
printString("Hi there");                    // ошибка: передача по ссылке
printString(String("Hi there"));            // OK: объект создается
int sz = String("Hi there").getSize();      // объект создается
```

Приведение может быть неявным (без использования оператора приведения), если компилятор соответствует требуемому типу, как в операторе присваивания:

```
String s;
s = "Hi there" ;          // то же, что и s = String("Hi there");
```

Во всех этих случаях в стеке создается неименованный объект String и вызывается конструктор преобразования. Затем объект удаляется. C++ не определяет точный момент для уничтожения объекта. Разработчик компилятора должен убедиться, что объект существует сразу же после его использования и исчезает до того, как закрывается текущая область действия.

Предполагается, что параметр конструктора преобразования должен иметь тип, принадлежащий одному из элементов данных класса. Например, у конструктора преобразования для класса String есть параметр в виде массива символов (указателя символов), а у класса String — элемент данных — указатель символов. Однако совсем не обязательно должно быть именно так.

Проектировщик класса может применить преобразование из любого типа, которое он посчитает подходящим. Например, класс Account из листинга 16.1 (или листинга 16.2) может включать в себя конструктор преобразований с параметром типа String, даже если в классе Account отсутствует элемент данных String. Конструктор имеет следующий вид.

```
Account(String& s)                  // преобразование (изменяется String)
{ char* p = s.reset();              // установить указатель на массив
  owner = new char[strlen(p)+1];    // выделение памяти из динамически
                                    // распределяемой области
  if (owner ==0) { cout < "\nOut of memory\n"; exit(0); }
  strcpy(owner, p);                 // инициализация полей данных
  balance = 0; }                    // значение по умолчанию для нового счета
```

Теперь объект String может использоваться в любом месте, где ожидается объект Account. Несмотря на то, что объект Account с нулевым балансом совершенно бесполезен, наиболее подходящим использованием этого конструктора является создание объектов Account, когда данные владельца представляются как объект String, а не как символьный массив.

```
String owner("Smith") ;
Account a(owner);           // создание и инициализация
a += 500;                   // использование объекта Account
```

Конструкторы преобразований позволяют проектировщику класса явно указать, какие типы могут использоваться там, где ожидаются значения указанного класса. Обратите внимание, что эти конструкторы реализуют приведение объектов, а не указателей или ссылок, которые всегда допустимы в C++. Приведение типов, реализованное конструкторами преобразования, может быть явным (если компилятор не может определить из контекста, над каким классом выполнять преобразование) или даже неявным (если цель преобразования ясна из контекста).

Конструкторы преобразований ослабляют строгий контроль типов. Они устраняют защиту компилятора от небрежного выполнения преобразования. Однако они очень популярны в C++, потому что допускают большую гибкость при написании программы.

Второй механизм определения возможных преобразований объектов — использование операций преобразования. Операция преобразования представляет собой перегруженную операцию, имя которой является именем типа цели. Операция преобразования подчиняется общим правилам для перегруженных операций. Однако ее синтаксис необычен.

Подобно конструкторам и деструкторам, она не должна иметь возвращаемый тип. Подобно деструкторам, у нее не должно быть параметров. В отличие от конструкторов и деструкторов операция преобразования должна возвращать значение. Тип этого значения будет соответствовать типу, в котором осуществляется преобразование (т. е. типу, используемому в имени операции). Операция преобразования в целый тип для класса String:

```
String::operator int() const        // отсутствуют изменения в объекте String
{ return size; }                    // возвращается только значение, не тип
```

Обычно возвращаемым значением является значение соответствующего типа одного из полей данных класса. Если у класса имеется более одного элемента данных этого типа, то проектировщик класса должен решить, какое из них лучше всего использовать в преобразовании. Если вы не знаете, какое поле выбрать, не стоит беспокоиться. Вспомните, что все операции используются для удобства синтаксиса клиентской программы.

В зависимости от обстоятельств, класс может содержать более одной операции преобразования. Так выглядит операция преобразования символьного указателя для класса String, который может заменить метод reset().

```
String::operator char* () const     // объект не изменяется
{ return str; }                     // возвращает указатель в начало
```

Упростите клиентскую программу (конструктор преобразований Account), используя две операции преобразования String.

```
Account(const String& s)
{ int len = (int)s;                  // получить размер строки
  owner = new char[len+1];           // выделить память в динамически
                                     // распределяемой области
  if (owner == 0) { cout << "\nOut of memory\n"; exit(0); }
  strcpy(owner, (char*)s);           // инициализировать поля данных
  balance = 0; }
```

Здесь явное приведение помогает программисту, осуществляющему сопровождение, понять поток значений в функции. Однако операции явного приведения не являются обязательными. Если компилятор без труда определяет требуемый тип, то явное приведение можно опустить, будет осуществляться неявное приведение.

```
Account(const String& s)
{ int len = s;                      // неявное приведение к целому типу
  owner = new char[len+1];          // выделение памяти в динамически
                                    // распределяемой области
  if (owner == 0) { cout < "\nOut of memory\n"; exit(0); }
  strcpy(owner, s);                 // неявное приведение к символьному массиву
  balance = 0; }
```

Объект String может использоваться в любом месте, где ожидается значение целого типа или символьного массива. Не забывайте, что под прикрытием программы C++ явное или неявное приведение является сообщением: вызовом

функции перегруженных операций преобразования. Вот как выглядят для компилятора обе версии конструктора.

```
Account(const String& s)
{ int len = s.operator int();              // вызов оператора
  owner = new char[len+1];                 // выделение памяти в динамически
                                           // распределяемой области
  if (owner == 0) { cout << "\nOut of memory\n"; exit(0); }
  strcpy(owner, s.operator char*());       // вызов оператора
  balance =0; }
```

В большинстве случаев операции преобразования используются для извлечения значения одного из полей объекта. Но это не является собственным ограничением. Операции преобразования используются проектировщиком класса для указания типа преобразований объектов класса. Все, что проектировщик определяет как допустимое преобразование, выполняется. Например, класс Account может поддерживать две операции преобразования в double и в String, даже если в классе Account не содержится элемент данных String.

```
Account::operator double () const      // объект не изменяется
{ return balance; }                    // возвращается значение двойной длины

Account::operator String () const      // создается объект String
{ return owner; }                      // неявное преобразование
```

Обратите внимание, что во второй операции преобразования выполняется неявное преобразование в класс String. Объект String создается и возвращается для использования в клиентской программе. Он автоматически уничтожается в области видимости клиента. Заметьте, что не сказано "когда клиентская часть прекращает выполнение", поскольку время уничтожения точно не определено. Использование ссылки в имени операции было бы синтаксически неправильно, поскольку все ссылки в C++ должны быть константами.

```
Account::operator String& () const     // синтаксическая ошибка
{ return owner; }                      // неявное преобразование
```

Чтобы устранить эту проблему, можно определить имя операции как ссылку String типа константа.

```
Account::operator const String& () const      // не синтаксическая ошибка
{ return owner; }                             // не очень хорошая идея
```

Однако возвращаемая ссылка является ссылкой на неименованный временный объект, который может быть уничтожен в любой момент, когда пожелает компилятор. В результате клиентская программа может получить недействительную ссылку. Это плохая практика программирования.

При наличии таких преобразований клиентская программа может преобразовать объект String в целое значение и в символьный указатель (с помощью операций преобразования), и в объект Account (используя конструктор преобразования Account). Она может преобразовать объект Account в значение двойной длины и в значение String (с помощью операций преобразования). Кроме того, массив символов можно преобразовать в объект String, используя конструктор преобразования String.

В листинге 16.6 показаны эти преобразования. Объект String обрабатывается клиентской программой, как если бы он был символьным массивом. Объект Account обрабатывается клиентской программой, как если бы он был значением двойной длины и значением String (см. рис. 16.6).

```
Name: Smith
Balance: 700
Credit limit: 1400
```

Рис. 16.6. *Вывод для программы из листинга 16.6*

Если в указанном контексте может использоваться несколько типов, то компилятор должен знать, какой из них применять. В качестве подсказки может использоваться приведение типов. В операторах вывода значение

любого типа может быть допустимым значением вывода. Явное приведение необходимо, если возможно преобразование более чем в один тип. Например, значение String может быть преобразовано в целое значение и в символьный массив. Компилятору требуется указать, что именно предполагал программист.

Листинг 16.6. Примеры использования конструкторов преобразования и операций преобразования

```
#include <iostream>
using namespace std;

class String {
   int size;                        // размер строки
   char *str;                       // начало внутренней строки
   void set (const char* s);        // размещение закрытой строки
public
   String (const char* s = "")      // по умолчанию и преобразование
     { set(s); }
   String (const String &s)         // конструктор копирования
     { set(s.str); )
   ~String()                        // деструктор
   { delete [ ] str; }
   String& operator = (const String& s);       // присваивание
   operator int() const;            // длина текущей строки
   operator char* () const;         // возвращение указателя в начало
} ;

void String::set(const char* s)
   { size = strlen(s);              // оценка размера
     str = new char[size + 1];      // запрос памяти в динамически распределяемой области
     if (str == 0) { cout << "Out of memory\n"; exit(0); }
     strcpy(str,s); }       // копирование клиентских данных в динамически распределяемую область

String& String::operator = (const String& s)
{  if (this == &s) return *this;   // ничего не делается, если самоприсваивание
   delete [ ] str;                 // возвращение существующей памяти
   set(s.str);                     // выделить/присвоить новую память
   return *this; }                 // для поддержки цепочечного присваивания

String::operator int() const       // изменения объекта String отсутствуют
{ return size; }

String::operator char* () const    // объект не изменяется
{ return str; )                    // возвращение указателя в начало

class Account {
protected:                         // базовый класс иерархии
   double balance;                 // защищенные данные
   char * owner;
public:

   Account(const char* name, double initBalance)      // общий
   { owner = new char[strlen(name)+1];       // выделение памяти для динамически
                                             // распределяемой области
     if (owner == 0) { cout << "\nOut of memory\n"; exit(0); }
     strcpy(owner, name);          // инициализация памяти динамически распределяемой области
     balance = initBalance; }

   Account(const String& s)
   { int len = s;                  // получение размера строки
     owner = new char[len+1];      // выделение памяти для динамически распределяемой области
```

```
    if (owner == 0) { cout << "\nOut of memory\n"; exit(0); }
    strcpy(owner, s);                          // инициализация полей данных
    balance = 0; }

  operator double () const                     // объект не изменяется
  { return balance; )

  operator String () const                     // создать объект String
  { return owner; }                            // неявное преобразование

  void operator -= (double amount)
  { balance -= amount; }                       // выбор из стека ответственности

  void operator += (double amount)
  { balance += amount; }                       // безусловное приращение
} ;

int main()
{
  String owner("Smith");                       // конструктор преобразования
  Account a(owner);                            // конструктор преобразования
  a += 500; a -=200; a += 400;                 // перегруженные операции
  String s = a;                                // обработать как значение String
  double limit = 2 * a;                        // обработать как значение двойной длины
  cout << "Name: " << (char *)s << endl;       // явное преобразование
  cout << "Balance: " <<(double)a <<endl;      // явное преобразование
  cout << "Credit limit: " << limit << endl;
  return 0;
  }
```

Если указанное преобразование не находится, то компилятор ищет встроенное преобразование (среди числовых типов), чтобы сделать возможным разрешение вызова. Рассмотрим оператор:

```
cout << "Balance: " <<(float)a <<endl;        // явное преобразование
```

Класс Account не обеспечивает операцию преобразования в тип с плавающей точкой (float). Однако это не означает, что приведенная выше строка содержит ошибку. Поскольку преобразование значений double типу float доступно как встроенное преобразование, компилятор переводит значение Account к типу double, а затем значение double к типу float. Компилятор не может добавить более одного встроенного преобразования float к преобразованию, определенному программистом. Компилятор не может присоединить в цепочке к преобразованию, определенному программистом, более одного встроенного преобразования. Однако в цепочке можно соединить встроенное и определенное программистом преобразование.

Операции, возвращающие компонент массива по индексу, и операции вызова функции

Эти две операции являются бинарными. Однако они отличаются от других бинарных операций C++ способом преобразования вызова операции клиентской части в бинарное выражение. Для других перегруженных операций цель сообщения используется как левый операнд в выражении с операцией (из имени метода вставленной между правым и левым операндом и параметром вызова функции

используемым как правый операнд выражения. В операциях, возвращающих компонент массива по индексу, и операциях вызова функции это не так.

Другое отличие состоит в том, что перегруженные операции могут быть реализованы только как компоненты класса. Невозможно реализовать их как глобальные функции, не являющиеся членами. Таким образом, становится легче осуществить контекстный анализ компилятора.

Операции, возвращающие компонент массива по индексу

В идеальном случае вид выражения перегруженной операции индексирования (операции, возвращающей компонент массива по индексу) должен быть таким же, как и синтаксический вид встроенной операции индексирования. Имя переменной добавляется с индексом, заключенным в скобки. Например, `s[i]` должно интерпретироваться как индекс `i`, применяемый к объекту (переменной) `s`.

Значение этой операции может быть произвольным. Большинство программистов и все библиотеки C++ интерпретируют такое выражение как извлечение значения `i`-ого компонента объекта `s`. Другой популярной интерпретацией является присвоение значения `i`-ому компоненту объекта `s`.

В обоих случаях предполагается, что объект `s` является контейнером, который содержит массив, или связанный список, или другой соответствующий набор компонентов, а выражение `s[i]` ссылается на значение `i`-го компонента в контейнере.

В качестве простого примера контейнерного класса рассмотрим упрощенный вариант класса `Array`. Этот контейнерный класс подобен классу из листинга 16.6. Компоненты контейнера имеют тип `int`, а не `char`.

Класс `Array` устраняет два недостатка встроенных массивов C++: переполнение массива и неверные значения индекса. Первая проблема решается с помощью размещения компонентов в динамически распределяемой области памяти. Для защиты целостности программы класс `Array` должен обеспечить конструктор копирования, деструктор и перегруженную операцию присваивания.

Вторая проблема с неверным индексом решается посредством функций-членов `getInt()` и `setInt()`, осуществляющих доступ к внутренней памяти `Array` от имени клиентской программы.

```
Class Array {
public:
   int size;                          // количество действительных компонентов
   int *ptr;                          // указатель на массив компонентов
   void set(const int* a, int n);     // выделить/инициализировать память
                                      // динамически распределяемой области
public:
   Array (const int* a,int n);        // общий конструктор
   Array (const Array &s);            // конструктор копирования
  ~Array();                           // освобождение памяти динамически
                                      // распределяемой области
   Array& operator = (const Array& a);     // копирование массива в другой
   int getSize() const;
   int getInt(int i) const;           // возвращение i-го компонента
   void setInt(int i, int x);         // установка int x в позицию i
} ;
```

Хотелось бы напомнить, что `i`-положение фактически означает `(i+1)` положение, т. е. первому компоненту соответствует индекс 0, второму компоненту — индекс 1 и т. д.

Следовательно, функция-член `getInt()` возвращает целое для индекса `i` массива внутренней динамически распределяемой области памяти. Поскольку `getInt()`

вызывается как функция, то дополнительно можно сделать некоторые полезные вещи. Рекомендуем проверить допустимость индекса относительно границ строки.

```
int Array::getInt (int i) const     // объект не изменяется
{ if (i < 0 || i >= size)           // индекс выходит за границы
    return ptr[size-1];             // возвращение к последнему компоненту
  return ptr[i]; }                  // допустимый индекс: возвращаемое значение
```

С помощью этой функции клиентская программа может реализовать итеративные алгоритмы, подобные алгоритмам, используемым для встроенных массивов C++. Однако они являются более безопасными, поскольку не могут осуществлять доступ к областям памяти за границами массива.

```
void printArray(const Array& a)
{ int size = a.getSize();              // получение размера массива
  for (int i=0; i < size; i++)         // просмотр каждого компонента
  { cout << " " << a.getInt(i); }      // вывод следующего компонента
  cout << endl << endl; }
```

Единственная проблема, связанная с этой дополнительной функциональной возможностью в `getInt()`, состоит в том, что она немного замедляет выполнение. Именно желание избежать этого замедления стало причиной того, что в C и C++ первоначально не была введена проверка индекса. Однако большинство современных приложений не страдает от этого. Можно использовать более быструю версию функции, которая вообще не будет тратить время на проверку индексов и будет реализована в виде встроенного ассемблерного кода.

В подобной структуре предполагается выполнение в клиентской программе проверки возвращаемого значения. Если такая проверка в клиентской части необходима, то размер данных объекта `Array`, который может быть извлечен с использованием функции-члена `getSize()` (как в функции `printArray()`, выше или аналогичен выполняемому в листинге 16.6 для подобного контейнера, класса `String`. Это позволяет клиентской части явно выполнить проверку. Однако каждое решение по проектированию представляет собой компромисс. Передача обязанностей серверной программе и упрощение клиентской программы рассматриваются в качестве надежной практики программной инженерии.

Возвращение последнего значения в контейнер, когда не действителен индекс,— хорошее решение. Другая альтернатива — возврат специального сигнального значения, например нуля. Это позволит клиентской программе структурировать итерации вокруг объекта `Array`, прекращая их при обнаружении нулевого кода. Но этот подход работает, только когда нулевое значение компонента является недействительным с точки зрения приложения.

Рассмотрим метод `setInt()`.

```
void Array::setInt(int i,int x)     // модификация объекта Array
{ if (i < 0 || i >= size)           // проверка допустимости индекса
    return;                         // выход, если вне границ
  ptr[i] = x; }                   // действительный индекс: присвоить значение
```

Можно доказать, что проверка границы для этой функции еще более важна, чем для метода `getInt()`. В `getInt()` существует риск внесения неверных данных в клиентскую программу, а это можно обнаружить во время отладки и тестирования. В `setInt()` риск состоит в искажении информации в памяти. В этом случае вы не сможете заблаговременно обнаружить ошибки.

Переведем версии двух функций, позволяющих клиентской программе работать с индексами, которые изменяются от 1 до границы массива.

```
int Array: : getInt (int i) const     // объект не изменяется
{ if (i < 1 | | i > size)             // индекс выходит за границы
    return ptr[size];                 // возвращение к последнему компоненту
  return ptr[i-1]; }                // допустимый индекс: возвращаемое значение
```

```
void Array::setInt(int i,int x)      // модификация объекта Array
{ if (i < 1 || i > size)             // проверка допустимости индекса
    return;                          // выход, если вне границ
  ptr[i-1 ] = x; }                 // действительный индекс: присвоить значение
```

Приведем вариант printArray(), который использует преимущество этой схемы. Теперь удобнее применять обычные преобразования и не требуется писать программы, подобные следующей:

```
void printArray(const Array& a)
{ int size = a.getSize();              // получение размера массива
  for (int i=1; i <= size; i++)        // выполнение от 1 до size
  { cout << " " << a. getInt (i); }    // вывод на печать следующего компонента
  cout << endl << endl; }
```

В листинге 16.7 представлена реализация класса Array с клиентской программой драйвера. Вывод программы показан на рис. 16.7.

В данном примере функции set(), конструкторы, деструктор и оператор присваивания подобны функциям-членам класса String из листинга 16.6. Основное отличие состоит в том, что функции String используют в своих циклах завершающий нуль, а функции Array — некоторое количество компонентов в контейнере.

```
1 3 5 7 11 13 17 19
2 6 10 14 22 26 34 38
```

Рис. 16.7. *Вывод программы из листинга 16.7*

Листинг 16.7. Использование класса Array как контейнера для целых компонентов

```
#include <iostream>
using namespace std;

class Array {
public:
   int size;                          // количество допустимых компонентов
   int *ptr;                          // указатель на массив компонентов
   void set(const int* a, int n);     // выделение/инициализация памяти
                                      // динамически распределяемой области
public:
   Array (const int* a,int n);        // общий конструктор
   Array (const Array &s);            // конструктор копирования
  ~Array();                           // возврат памяти в "кучу"
   Array& operator = (const Array& a); // копирование массива в другой массив
   int getSize() const;
   int getInt(int i) const;           // возвращение i-го компонента
   void setInt(int i, int x);         // установка int x в позицию i
} ;

void Array::set(const int* a, int n)
{  size = n;                          // определение размера массива
   ptr = new int[size];               // запрос памяти динамически распределяемой области
   if (ptr == 0) { cout << "Out of memory\n" exit(0); }
   for (int i=0; i < size; i++)
       ptr[i] = a[i]; }               // копирование клиентских данных в "кучу"

Array::Array (const int* a, int n)    // общее
   { set(a,n); }

Array::Array (const Array &a)         // копирование конструктора
   { set(a.ptr,a.size); }

Array::~Array()                       // деструктор
   { delete [ ] ptr; }
```

```
Array& Array::operator = (const Array& a)
{  if (this == &a) return *this;          // ничего, если допускает самоприсваивание
   delete [ ] ptr;                        // возвращение существующей памяти
   set(a.ptr,a.size);                     // выделение/установка новой памяти
   return *this; }                        // поддержка цепочечного присваивания

int Array::getSize() const                // получение размера массива
{ return size; }

int Array::getInt (int i) const           // объект не изменяется
{ if (i < 0 || i >= size)                 // индекс выходит за границы
     return ptr[size-1];                  // возвращение последнего компонента
  return ptr[i]; }                        // допустимый индекс: возвращение значения

void Array::setInt(int i,int x)           // модификация объекта Array
{ if (i < 0 || i >= size)                 // проверка допустимости индекса
     return;                              // выход, если вне границ
  ptr[i] = x; }                           // действительный индекс: присвоить значение

int main()
{
   int arr[] = { 1,3,5,7,11,13,17,19 } ;  // данные для обработки
   Array a(arr, 8);                       // создать объект
   int size = a.getSize();                // получить размер массива
   for (int i=0; i < size; i++)           // просмотр каждого компонента
   { cout << " " << a.getInt(i); }        // вывод на печать следующего компонента
   cout << endl << endl;
   for (int j=0; j < size; j++)           // повторный просмотр массива
   { int x = a.getInt (j);                // получение следующего компонента
     a.setInt(j, 2*x); }                  // корректировка значения
   for (int k = 0; k < size; k++)
   { cout << " " << a.getInt(k); }        // вывод на печать скорректированного массива
   cout << endl;
   return 0;
}
```

Окончательно готовый класс Array также должен поддерживать добавление новых компонентов в конец и в середину массива, удаление компонентов, сравнение компонентов, тестирование на наличие допустимых данных и т. п.

Как упоминалось выше, синтаксис использования метода getInt() удобен. Синтаксис применения метода setInt() более громоздкий.

```
for (int j=0; j < size; j++)        // повторный просмотр массива
  { int x = a.getInt(j);            // получение следующего компонента
    a.setInt(j, 2*x); }             // корректировка значения
```

Итак, мы просмотрели все компоненты массива. Значение каждого компонента увеличено вдвое. Синтаксис изменения отличается от синтаксиса доступа к компонентам. Между тем встроенные массивы C++ используют тот же самый синтаксис доступа к компонентам массива (например, x = a[j]) и для присваивания значений компонентам массива (например, a[j] = 2*x).

Хорошо было бы структурировать клиентскую программу для корректировки значений в контейнере одновременно с доступом к значениям.

```
   for (int j=0; j < size; j++)       // повторный просмотр массива
     { int x = a.getInt(j);           // получение следующего компонента
//     a.setInt(j, 2*x); }            // корректировка значения
     a.setInt(j) = 2 * x; }           // корректировка значения
```

В традиционном программировании это невозможно — возвращаемое значение функции не может использоваться в левой части присваивания. C++ обеспечивает такую возможность, если функция возвращает ссылку на значение, а не само значение. Ссылка должна быть действительной и не должна исчезать при завершении функции.

В главе 7 обсуждались возможности, которые открывает для написания краткой и выразительной клиентской программы возвращение ссылки из функций. Устраним параметр-значение для интерфейса setInt() и изменим возвращаемый тип setInt() из целого значения в целый указатель.

```
int& Array::setInt(int i)        // изменение объекта Array
{ if (i < 0 || i >= size)        // проверка допустимости индекса
    return ptr[size-1];          // возвращение последнего компонента
  return ptr[i]; }               // действительный индекс: присвоить значение
```

Эта функция поддерживает клиентский цикл, приведенный выше: он возвращает ссылку на целое значение, а в цикле значение присваивается по адресу, на который указывает ссылка. Критическим компонентом этой схемы является то, что ссылка указывает не на локальное значение, которое исчезало бы, когда функция setInt() завершается. Ссылка указывает на компонент массива, который имелся до вызова setInt() и будет существовать после завершения setInt().

Сравним getInt() и новую версию setInt(). Видно, что их реализации одинаковые. Требуются ли клиентской программе обе функции? Между ними существуют два различия, касающихся интерфейса функции. Возвращенное значение getInt() является значением, а не ссылкой. Это не серьезная проблема. Изменим возвращаемое значение getInt() на ссылку на целое число.

```
int& Array::getInt(int i) const      // объект не изменяется
{ if (i < 0 || i >= size)            // индекс вне границ
    return ptr[size-1];              // возвращение последнего компонента
  return ptr[i]; }                 // действительный индекс: присвоить значение
```

С этой функцией клиентская программа в листинге 16.7 будет работать, как и прежде.

```
for (int i=0; i < size; i++)         // просмотр каждого компонента
{ cout << " " << a.getInt(i); }      // OK, если возвращается ссылка
cout << endl << endl;
for (int j=0; j < size; j++)         // повторный просмотр всех компонентов
  { int x = a.getInt(j);             // OK, если возвращается ссылка
    a.setInt(j) = 2 * x; }           // OK, если возвращается ссылка
```

Второе отличие состоит в том, что getInt() не изменяет состояния обрабатываемого объекта, а помечается как константа. С другой стороны, setInt() меняет состояние объекта, отправляемого как сообщение, следовательно, оно не указывается как константа.

Это типичная ошибка многих программистов C++, сталкивающихся с использованием модификаторов const. Функция setInt() изменяет состояние динамически распределяемой области памяти, которая принадлежит объекту цели. Но эта память не является частью объекта — она только принадлежит ему. Элементы данных представляет собой часть объекта, а не памяти динамически распределяемой области. Функция setInt() не изменяет элементы данных объекта цели. Это одна из тех концепций, которую всегда должен помнить программист C++.

Функция-член setInt() спроектирована неверно. Ее необходимо пометить как const, потому что она не изменяет состояния своего объекта цели.

```
int& Array::setInt(int i) const      // объект Array не изменяется
{ if (i < 0 || i >= size)            // проверка действительности индекса
    return ptr[size-1];              // возвращение последнего компонента
  return ptr[i]; }                 // действительный индекс: присвоить значение
```

Несмотря на комментарии, показывающие, что объект изменяется, setInt() в листинге 16.7 не меняет свой объект цели. Вспомните историю о кирпиче, рассказанную в главе 8. Не забывайте о модификаторах const.

Теперь, когда обе функции getInt() и setInt() выглядят одинаково, можно исключить одну из них. В листинге 16.8 показан вариант программы из листинга 16.7, в котором используется только одна функция getInt(). Вывод для этого примера такой же, как и для листинга 16.7.

Листинг 16.8. Использование одной функции-члена для получения и установки данных Array

```
#include <iostream>
using namespace std;

class Array {
public:
   int size;                              // количество допустимых компонентов
   int *ptr;                              // указатель на массив целых величин
   void set(const int* a, int n);         // выделить/инициализировать память
                                          // динамически распределяемой области
public:
   Array (const int* a, int n);           // общий конструктор
   Array (const Array &s);                // конструктор копирования
  ~Array();                               // возврат памяти динамически распределяемой области
   Array& operator = (const Array& a);    // копирование массива
   int getSize() const;
   int& getInt(int i) const;              // получить/установить значение в позицию i
} ;

void Array::set(const int* a, int n)
{  size = n;                              // оценить размер массива
   ptr = new int[size];                   // запросить память динамически
                                          // распределяемой области
   if (ptr ==0) { cout < "Out of memory\n"; exit(0); }
   for (int i=0; i < size; i++)
        ptr[i] = a[i]; }                  // скопировать клиентские данные в "кучу"

Array::Array(const int* a, int n)         // общий конструктор
   { set(a,n); }

Array::Array (const Array &a)             // конструктор копирования
   { set(a.ptr, a.size); }

Array::~Array()                           // деструктор
   { delete [ ] ptr; }

Array& Array::operator = (const Array& a)
{  if (this == &a) return *this;          // ничего, если обеспечивает самоприсваивание
   delete [ ] ptr;                        // возвращение существующей памяти
   set(a.ptr,a.size);                     // выделение/установка новой памяти
   return *this; }                        // для поддержки цепочечного присваивания

int Array::getSize() const                // получить размер массива
{ return size; }

int& Array::getInt(int i) const           // объект Array не изменяется
{ if (i < 0 || i >= size)                 // проверка допустимости индекса
     return ptr[size-1];                  // выход, если вне границ
  return ptr[i]; }                        // действительный индекс: присвоить значение
```

```
int main()
{
  int arr[] = { 1,3,5,7,11,13,17,19 } ;    // данные для обработки
  Array a(arr, 8);                         // создать объект
  int size = a.getSize();                  // получить размер массива
  for (int i=0; i < size; i++)             // просмотр каждого компонента
  { cout << " " << a.getInt(i); }          // вывод на печать следующего компонента
  cout << endl << endl;
  for (int j=0; j < size; j++)             // повторный просмотр массива
  { int x = a.getInt(j);                   // получить следующий компонент
    a.getInt(j) = 2*x; }                   // корректировка значения
  for (int k = 0; k < size; k++)
  { cout << " " << a.getInt(k); }          // вывод на печать скорректированного массива
  cout << endl;
  return 0 ;
  }
```

Следующим шагом является замена функции-члена `getInt()` на перегруженную операцию индексирования. Изменение самой функции очень простое. Берется функция, вырезается ее имя `getInt`, перемещается в зарезервированное слово `operator` и добавляется символ для операции (в данном случае `[]`).

```
// int& Array::getInt(int i) const       // объект Array не изменяется
int& Array::operator [](int i) const     // заголовок операции
{ if (i < 0 || i >= size)                // проверка допустимости индекса
    return ptr[size-1] ;                 // выход, если вне границ
  return ptr[i]; }              // действительный индекс: определить ссылку
```

Подобные изменения должны быть сделаны в клиентской программе — имя функции-члена теперь будет `operator[]`, а не `getInt`.

```
int size = a.getSize();                  // получить размер массива
for (int i=0; i < size; i++)             // просмотр каждого компонента
{ cout << " " << a.operator![](i); }     // вывод на печать
                                         // следующего компонента
cout << endl << endl;
for (int j=0; j < size; j++)             // повторный просмотр массива
{ int x = a.operator[](j);               // получить следующий компонент
  a.operator[](j) = 2*x; }               // корректировка значения
for (int k = 0; k < size; k++)
{ cout << " " << a. operator[] (k); }    // вывод на печать
                                         // скорректированного массива
cout << endl;
```

Весь путь от первой реализации в листинге 16.7 не был проделан только для того, чтобы остановиться на этом. Синтаксис с вызовом функции следует заменить на синтаксис с выражением. Однако интерпретация `operator[]`, как и любой другой операции, дает в результате громоздкую программу. Как, например, интерпретировать `operator+`? Цель сообщения используется как первый операнд, затем как символ из оператора, например `+`, и потом как второй операнд.

```
a.operator+(b);                          // то же, что и a + b;
```

Если выполнить это же с операцией индексирования, получится что-то нечитаемое.

```
cout << " " << a.operator[ ](i);         // то же, что и a[]i
```

Чтобы функция операции индексирования не противоречила использованию встроенной операции индексирования, C++ отбрасывает специальную часть. Компилятору дается указание разрешить отклонение от общего правила. В листинге 16.9 приведен этот пример с использованием перегруженной операции индексирования.

Листинг 16.9. Использование перегруженной операции индексирования для получения и определения значения данных `Array`

```
#include <iostream>
using namespace std;

class Array {
public:
   int size;                          // количество действительных компонентов
   int *ptr;                          // указатель на массив целых значений
   void set(const int* a, int n) ;    // выделить/инициализировать память
                                      // динамически распределяемой области
public:
   Array (const int* a, int n);       // общий конструктор
   Array (const Array &s);            // конструктор копирования
  ~Array();                           // возвращение памяти динамически распределяемой области
   Array& operator = (const Array& a);          // копировать массив
   int getSize() const;
   int& operator [ ] (int i);         // получить/установить значение в положение i
} ;

void Array::set(const int* a, int n)
{  size = n;                          // оценить размер массива
   ptr = new int[size];               // запросить память из динамически распределяемой области
   if (ptr == 0) { cout << "Out of memory\n"; exit(0); }
   for (int i=0; i < size; i++)
       ptr[i] = a[i]; }               // скопировать клиентские данные в "кучу"

Array::Array(const int* a, int n)     //общий конструктор
   { set(a,n); }

Array::Array (const Array &a)         // конструктор копирования
   { set(a.ptr,a.size); }

Array::~Array()                       // деструктор
   { delete [ ] ptr; }

Array& Array::operator = (const Array& a)
{  if (this == &a) return *this;      // ничего, если обеспечивается самоприсваивание
   delete [ ] ptr;                    // возвращение существующей памяти
   set(a.ptr,a.size);                 // выделение/установка новой памяти
   return *this; }                    // для поддержки цепочечного присваивания

int Array::getSize() const            // получить размер массива
{ return size; }

int& Array::operator [](int i)        // объект Array не изменяется
{  if (i < 0 || i >= size)            // проверка допустимости индекса
       return ptr[size-1];            // выход, если вне границ
   return ptr[i]; }                   // действительный индекс: установить значение

int main ()
{  int arr[] = { 1,3,5,7,11,13,17,19 } ;
   Array a(arr, 8);                   // данные для обработки
   int size = a.getSize();            // создать объект
```

```
  for (int i=0; i < size; i++)          // получить размер массива
//{ cout << " " <<a. operator[](i); }   // альтернативный синтаксис
  { cout << " " << a[i]; }              // вывод на печать следующего компонента
  cout << endl << endl;
  for (int j=0; j < size; j++)          // повторный просмотр всего массива
     { int x = a[j];                    // специальное присваивание
//     { int x = a.operator[](j);       // альтернативный синтаксис
       a[j] = 2*x; }                    // специальное присваивание
  for (int k = 0; k < size; k++)
  { cout << " " << a[k]; }              // вывод на печать скорректированного массива
  cout << endl;
  return 0;
}
```

Не совсем понятно, что улучшилось в этом варианте по сравнению с оригиналом из листинга 16.7. Однако синтаксис операции полезен для рассмотрения вопросов, связанных с возвратом ссылки из функции, а не для возвращения значения и использования модификаторов const.

Операция вызова функции

Операция вызова функции (две скобки рассматриваются как операция в C++) также может использоваться для осуществления доступа или определения значений компонентов объекта контейнерного класса. Операция применяется, когда контейнер рассматривает структуру динамически распределяемой области памяти как двумерный массив.

Причина использования операции вызова функции вместо операции индексирования заключается в том, что для двумерного массива C++ использует две объединенные операции индексирования, например `m[i][j]`. Применение синтаксиса обычного программирования с одной операцией индексирования, например `m[i,j]`, сделало бы индекс тернарной операцией. (В этом случае ее операндами являются массив `m` и индексы `i` и `j`.) В многомерных массивах может быть больше двух индексов.

Разработчики C и C++ понимали, что все будет в порядке, если разрешить операции плюса изменить свою четность — допуская как унарный плюс, так и бинарный плюс. Но подобное разрешение не было сделано для операции индексирования. Это бинарная операция, и у нее не может быть более двух операндов.

Вместо операции индексирования можно воспользоваться операцией вызова функции. Ее преимущество в этой ситуации состоит в том, что она может иметь любое количество параметров.

Для примера рассмотрим класс `Matrix`, который реализует квадратную матрицу. Клиентская программа обрабатывает компоненты матрицы, определяет два индекса — один для строки и один для столбца матрицы. Объекты матрицы могут создаваться, передаваться как параметры функции и присваиваться друг другу. Реализация будет основываться на динамически выделенном линейном массиве, размер которого зависит от размера квадратной матрицы.

Класс `Matrix` использует закрытую функцию `make()`, подобную функции `set()` из предыдущего примера, но она не выполняет инициализацию памяти динамически распределяемой области. Функция `make()` вызывается конструктором преобразования, конструктором копирования и перегруженным оператором присваивания.

```
class Matrix {
  int *cells;              // массив в динамически распределяемой области
                           // памяти для размещения матрицы
  int size;                // количество строк и столбцов
```

```
  int* make(int size)                    // закрытая функция-распределитель
  { int* p = new int [size * size];      // общее количество компонентов
    if (p == NULL) { cout << "Matrix too big\n"; exit(0); }
    return p; }                        // возврат указателя на память динамически
                                         // распределяемой области
public:
  Matrix (int sz) : size(sz)             // конструктор преобразования
  { cells = make(size); }                // память динамически распределяемой
                                         // области не инициализируется
  Matrix (const Matrix&: m) : size(m.size)
  { cells = make(size); }          // конструктор копирования: для безопасности
  Matrix& operator = (const Matrix m);          // оператор присваивания
  int getSize() const                    // размер стороны
  { return size; }
  int& get (int r, int c) const;         // доступ или модификация компонентов
  ~Matrix() { delete [] cells; }         // деструктор
} ;
```

Оператор присваивания освобождает существующую память динамически распределяемой области, выделяет новую память в динамически распределяемой области и копирует данные из параметра объекта Matrix в цель присваивания.

```
Matrix& Matrix::operator = (const Matrix& m)      // присваивание
{ if (this == &m) return *this;      // ничего, если обеспечивается
                                     // самоприсваивание
  delete [ ] cells;                  // возврат существующей памяти
  cells = make(m.size);              // выделение/установка памяти
  size = m.size;                     // установить размер матрицы
  for (int i=0; i<size*size; i++)    // копировать данные
       cells[i] = m.cells[i];
  return *this; }                    // для поддержки цепочечного присваивания
```

Функция get() объединяет обязанности функций getInt() и setInt() из предыдущего примера. Она использует координаты строки и столбца, переданные вызывающей операцией (начиная от нуля) для вычисления положения компонента матрицы в линейном массиве. Если координаты являются недопустимыми, она просто возвращает последний элемент массива. Если координаты правильные, она возвращает данные, сохраненные по указанным координатам.

```
int& Matrix::get (int r, int c) const
{ if (r<0 || c<0 || r>=size || c>=size)  // проверка правильности значения
    return cells[size*size-1];    // возвращение последнего элемента матрицы
  return cells[r*size + c]; }             // возвращение запрошенного элемента
```

Возвращение последнего элемента матрицы, если координаты строки или столбца выходят за границы матрицы, не самое лучшее решение. Другая возможность заключается в завершении выполнения или в генерации исключительной ситуации. Кроме того, можно возвратить некоторое значение, не используемое каким-либо иным образом в приложении. Например, максимальное целое значение max_int. Однако значение-константу невозможно вернуть по ссылке (если только не модифицировать его).

```
int& Matrix::get (int r, int c) const      // не очень хороший вариант
{ if (r<0 || c<0 || r>=size || c>=size)    // проверка правильности значения
    return MAX_INT;                        // не допускается возврат по ссылке
  return cells[r*size + c]; }             // возвращение запрошенного элемента
```

В листинге 16.10 показана программа, которая реализует класс Matrix с только что описанной функцией get(). Клиентская функция printMatrix() просматривает строки и столбцы матрицы и выводит на печать по очереди каждую строку. Обратите внимание на использование манипулятора setw(). К сожалению, включаемого файла <iostream> недостаточно для программы, использующей манипуляторы, и в заголовке файла требуется указать <iomanip>.

Листинг 16.10. Использование класса Matrix в качестве контейнера для квадратной матрицы

```
#include <iostream>
#include <iomanip>
using namespace std;

class Matrix {
  int *cells;                              // массив в динамически распределяемой
                                           // области памяти для размещения матрицы
  int size;                                // количество строк и столбцов
  int* make(int size)                      // закрытая функция-распределитель
  { int* p = new int [size * size];        // общее количество элементов
    if (p == NULL) { cout << "Matrix too big\n"; exit(0); }
    return p; }                            // возвращение указателя на динамически
                                           // распределяемую область памяти
public:
  Matrix (int sz) : size(sz)               // конструктор преобразования
  { cells = make(size); }                  // память динамически распределяемой
                                           // области не инициализируется
  Matrix (const Matrix& m): size(m.size)
  { cells = make(size); }                  // конструктор копирования: для безопасности
  Matrix& operator = (const Matrix& m);    // оператор присваивания
  int getSize() const                      // размер стороны
  { return size; }
  int& get (int r, int c) const;           // доступ или модификация компонента
  ~Matrix() { delete [] cells; }           // деструктор
  } ;

Matrix& Matrix::operator = (const Matrix& m)       // присваивание
{ if (this == &m) return *this;            // ничего, если обеспечивается самоприсваивание
  delete [ ] cells;                        // возвращение существующей памяти
  cells = make(m.size);                    // выделение/установка новой памяти
  size = m.size;                           // установка размера матрицы
  for (int i=0; i<size*size; i++)          // копирование данных
      cells[i] = m.cells[i];
  return *this; }                          // поддержка цепочечного присваивания

int& Matrix::get (int r, int c) const
  { if (r<0 || c<0 || r>=size || c>=size)  // проверка допустимости
       return cells[size*size-1];          // возвращение последнего элемента матрицы
    return cells[r*size + c]; }            // возвращение запрашиваемого элемента

void printMatrix(const Matrix& m)          // клиентская функция
{ int size = m.getSize();
  for (int i=0; i < size; i++)             // просмотр каждой строки
  { for (int j=0; j < size; j++)           // и каждого столбца
       cout <<setw(4) <<m.get(i,j);        // печать элемента m[i][j]
    cout << endl; }                        // конец текущей строки
  cout << endl; }                          // конец матрицы
```

```
int main()
{  cout << endl << endl;
   int i, j, n = 5; Matrix m1(n);              // объект Matrix
   for (i=0; i < n; i++)
       for (j=0; j < n; j++)                   // инициализация ячеек
           m1.get(i,j) = (i+1) * (j+1);        // m1[i][j] = (i+i)*(j+i);
   printMatrix(m1);                            // вывод на печать состояния матрицы
   for (i=0; i < n; i++)                       // занесение нулей по главной диагонали
       m1.get(i,i) = 0;                        // m1[i][i] = 0
   printMatrix(m1);                            // вывод на печать нового состояния
   cout <<"m[10][10] = " <<m1.get(10,10) <<endl;        // вне диапазона
   return 0;
   }
```

```
    1   2   3   4   5
    2   4   6   8  10
    3   6   9  12  15
    4   8  12  16  20
    5  10  15  20  25

    0   2   3   4   5
    2   0   6   8  10
    3   6   0  12  15
    4   8  12   0  20
    5  10  15  20   0

m[10][10] = 0
```

Рис. 16.8.
Вывод программы из листинга 16.10

Клиентская функция main() создает объект — квадратную матрицу и инициализирует каждый компонент значением, равным произведению его номера строки и номера столбца. В этом цикле функция main() использует возвращенное значение функции get() как l-значение. Затем main() вызывает printMatrix(), которая использует возвращенное значение функции get() как r-значение. Следом функция main() устанавливает компоненты диагонали главной матрицы в нуль (используя get() как l-значение) и выводит на печать матрицу. Наконец, main() пытается осуществить доступ к ячейке вне матрицы, и функция get() возвращает значение последнего элемента матрицы, которое было установлено в нуль. Результаты выполнения представлены на рис. 16.8.

Преобразование функции get() в перегруженную операцию вызова функции очень простое. Имя get заменяется зарезервированным словом operator и добавляются символы операции (две пустые скобки).

```
int& Matrix::operator() (int r, int c) const
  { if (r<0 || c<0 || r>=size || c>=size)    // проверка правильности значения
       return cells[size*size-1];  // возвращение последнего элемента матрицы
    return cells[r*size + c]; }     // возвращение запрошенного элемента
```

Эту функцию можно вызвать, используя синтаксис вызова функции как синоним для get().

```
void printMatrix(const Matrix& m)              // клиентская функция
{ int size = m.getSize();
  for (int i=0; i < size; i++)                 // просмотр каждой строки
  {  for (int j=0; j < size; j++)              // и каждого столбца
        cout <<setw(4) <<m.operator()(i,j);    // элемент m[i][j]
    cout << endl; }                            // конец текущей строки
  cout << endl; }                              // конец матрицы
```

Все перегруженные операции выглядят странно, пока к ним не привыкнешь. Преобразование синтаксиса вызова функции в синтаксис выражения также необычно. Формальное применение правил C++ даст в результате что-то подобное m()i,j. Вместо этого C++ разрешает записать это как m(i,j). Некоторым программистам C++ кажется, что подобный синтаксис не осуществляет доступ к компонентам матрицы. Однако многим программистам, работающим в научных областях, это напоминает то, что позволяет делать FORTRAN.

В листинге 16.11 показана полная версия программы из листинга 16.10, где вызовы функции get() заменяются на вызовы перегруженной операции вызова функции — operator()(). (Имя operator()() формируется так же, как и имя

любой другой функции.) Вывод программы соответствует выводу программы в листинге 16.10 (см. рис. 16.8).

Листинг 16.11. Использование класса `Matrix` с перегруженной операцией вызова функции

```
#include <iostream>
#include <iomanip>
using namespace std;

class Matrix {
  int *cells;                                  // массив динамически распределяемой
                                               // области памяти для размещения матрицы
  int size;                                    // количество строк и столбцов
  int* make(int size)                          // закрытая функция-распределитель
  { int* p = new int [size * size];            // общее количество компонентов
    if (p == NULL) { cout << "Matrix too big\n"; exit(0); }
    return p; }                                // возвращение указателя на динамически
                                               // распределяемую область памяти
public:
  Matrix (int sz) : size(sz)                   // конструктор преобразования
  { cells = make(size); }                      // память динамически распределяемой
                                               // области не инициализируется
  Matrix (const Matrix& m) : size(m.size)
  { cells = make(size); }                      // конструктор копирования: для безопасности
  Matrix& operator = (const Matrix& m);        // оператор присваивания
  int getSize() const                          // размер стороны
  { return size; }
  int& operator () (int r, int c) const;       // доступ или модификация
  ~Matrix() { delete [] cells; }               // деструктор
  } ;

Matrix& Matrix::operator = (const Matrix& m)   // присваивание
{ if (this == &m) return *this;                // ничего, если самоприсваивание
  delete [ ] cells;                            // возвращение существующей памяти
  cells = make(m.size);                        // выделение/установка новой памяти
  size = m.size;                               // установка размера матрицы
  for (int i=0; i<size*size; i++)              // копирование данных
      cells[i] = m.cells[i];
  return *this; }                              // для поддержки цепочечного присваивания

int& Matrix::operator () (int r, int c) const
  { if (r<0 || c<0 || r>=size || c>=size)      // проверка правильности значения
       return cells[size*size-1];              // возвращение последнего компонента матрицы
    return cells[r*size + c]; }                // возвращение запрошенной ячейки

void printMatrix(const Matrix& m)              // клиентская функция
{ int size = m.getSize();
  for (int i=0; i < size; i++)                 // просмотр каждой строки
  { for (int j=0; j < size; j++)               // и каждого столбца
       cout << setw(4) << m(i,j);              // вывод на печать элемента
    cout << endl; }                            // конец текущей строки
  cout << endl; }                              // конец матрицы

int main()
{ cout << endl << endl;
  int i, j, n = 5; Matrix m1(n);               // объект Matrix
  for (i=0; i < n; i++)
      for (j=0; j < n; j++)                    // инициализация элементов
          m1(i,j) = (i+1) * (j+1);             // m1[i][j] = (i+1)*(j+1);
```

```
printMatrix(m1);                          // вывод на печать состояния матрицы
for (i=0; i < n; i++)                     // занесение нулей в элементы главной диагонали
    m1(i,i) = 0;                          // m1[i][i] = 0
printMatrix(m1);                          // вывод на печать нового состояния
cout << "m[10][10] = " << m1(10,10) << endl;     // выход за границы
return 0;
}
```

Эта структура не поддерживает сложение, умножение, сравнение и другие полезные операции с матрицами. Ее назначение состоит только в том, чтобы продемонстрировать использование операции вызова функции.

Операции ввода/вывода

Стандартная библиотека C++ перегружает операцию ввода >> и операцию вывода << для всех встроенных классов. Очевидно, операции ничего не знают о классах, определенных программистом. Именно поэтому, когда требуется выполнить ввод или вывод данных объекта, необходимо делать это для каждого элемента данных объекта в отдельности.

Было бы прекрасно перегружать операции ввода/вывода также и для классов, определенных программистом. Инкапсулирование этих операций в перегруженные операции способствовало бы упрощению клиентской программы, исключение деталей нижнего уровня из клиентской программы и позволило бы передать некоторые обязанности из клиентской программы в серверные классы.

Перегрузка операции >>

Рассмотрим класс String из листинга 16.6, осуществляющий динамическое управление своей памятью и поддержку доступа клиентской программы к ее внутренним данным.

```
class String {
  int size;                                  // размер строки
  char *str;                                 // начало внутренней строки
  void set(const char* s);                   // выделение закрытой строки
public:
  String (const char* s = "")                // по умолчанию и преобразование
    { set(s); }
  String (const String &s)                   // конструктор копирования
    { set(s.str); }
  ~String()                                  // деструктор
    { delete [ ] str; }
  String& operator = (const String& s);      // присваивание
  operator int() const;                      // длина текущей строки
  operator char* () const;                   // возвращение указателя в начало
} ;
```

Было бы чудесно выполнить перегрузку операций ввода/вывода для этого класса так, чтобы клиентская программа могла использовать что-то подобное:

```
int main ()
{ String s;
  cout << "Enter customer name: ";
  cin >> s;                                  // принять имя
  cout << "The customer name is: ";
  cout << s << endl;                         // вывести на экран имя
  return 0; }
```

Интерфейс перегруженной операции ввода — это бинарный оператор, который воздействует на потоковый объект типа i (который поддерживает ввод для библиотечного объекта cin и дисковых файлов) и на объект типа String. Давайте выполним перегрузку операции >> для класса String.

```
void String::operator >>(istream& in)
{ char name[80];                    // локальная память для данных
  in >> name;                       // принять данные
  delete [] str;                    // возвращение существующей памяти
  set(name); }                      // выделение/инициализация новой памяти
```

Обратите внимание на то, что параметр функции передается по ссылке, а не по значению, поскольку он управляет своей памятью динамически. Заметьте, что это не ссылка на объект const,— вводимый объект изменяется в зависимости от операции ввода. Учитывайте, что функция не помечена как const, поскольку она изменяет состояние элементов данных объекта, освобождает существующую память динамически распределяемой области и устанавливает указатель на другую область памяти динамически распределяемой области памяти.

Это работает. Однако подобная операция имеет громоздкий интерфейс. В соответствии с правилами C++ для преобразования синтаксиса вызова функции в синтаксис выражения необходимо вызвать ее с объектом String как целью, а объектом istream как параметром.

```
s.operator >> (cin);                // эквивалентно s >> cin;
```

Желательно иметь специальное разрешение, подобное приведенному выше для операции индексирования и операции вызова функции. Но в данном случае оно отсутствует, и ни один программист не сможет использовать операцию ввода, если объект cin не будет операндом в левой части.

Если это не срабатывает, давайте спроектируем эту операцию как член класса istream. Класс istream является библиотечным классом.

Итак, эту операцию невозможно перегрузить как член класса istream, но можно (хотя и нет желания) перегрузить ее как член класса String. Что нам следует делать? В отсутствие всех других альтернатив нам не остается ничего другого, кроме перегрузки этой операции как глобальной функции.

```
void operator >> (String& s, istream& in)     // глобальная функция
{ char name[80];                    // локальная память для данных
  in >> name;                       // принять данные
  String temp(name);                // создать/инициализировать новый объект
  s = temp; }                       // копировать его в аргумент
```

Это не безупречно и довольно медленно. Во-первых, создается временный строковый объект, а затем он копируется в аргумент. Но это является частью внешнего ввода/вывода, поэтому никоим образом не влияет на производительность программы.

Вот как вызывается эта функция с использованием синтаксиса вызова функции: имя функции, первый параметр и второй параметр.

```
operator >> (s, cin);               // то же, что и s >> cin
```

Как можно видеть, приобретено не так уж и много. Проблема состоит в том, что объект String является первым операндом, а не вторым. Давайте изменим порядок параметров функции.

```
void operator >> (istream& in, String& s)     // глобальная функция
{ char name[80];                    // локальная память для данных
  in >> name;                       // принять данные
  String temp(name);                // создать/инициализировать новый объект
  s = temp; }                       // копировать его в аргумент
```

Вот это намного лучше — с объектом String как вторым параметром синтаксис выражения такой, как нам хотелось видеть.

```
operator >> (cin, s);              // то же, что и cin >> s;
```

На следующем этапе эту функцию нужно сделать "другом" класса String, чтобы не возиться с объектом цели, а перейти непосредственно к использованию его не общедоступной функции set() и элементов данных.

```
class String {
  int size;                                  // размер строки
  char *str;                                 // начало внутренней строки
  void set(const char* s);                   // выделение закрытой строки
public:
friend void operator >> (istream& in, String& s);
. . . } ;                                    // остальная часть класса String
```

Это почти безукоризненно. Однако с точки зрения интерпретации классов, определенных программистом, подобно встроенным классам, такая функция не оправдывает надежд. Для встроенных типов библиотека iostream поддерживает цепочечные операции.

```
double x, y;
cin >> x >> y;                  // то же, что и cin >> x; cin >> y;
```

Для нашего примера эта клиентская программа не работает — она генерирует синтаксическую ошибку.

```
String s; int qty;
cout << "Enter customer name and quantity: ";
cin >> s >> qty;                // ошибка: не цепочечные вызовы
```

Какой смысл последней строки в последнем фрагменте программы? Если вы рассмотрите определение функции operator >> (перегруженной для всех возможных типов), то увидите, что возвращаемый ей тип является ссылкой на тип istream. Данный цепочечный синтаксис возможен — никакое специальное разрешение не сделало бы этого.

Эта функция была вызвана с помощью объектов cin и s как параметров. Когда операция возвращает ссылку на объект istream, ему отправляется другая версия операции из библиотечного класса istream с переменной qty в качестве аргумента.

```
(operator>>(cin,s)).operator>>(qty);        // то же, что и cin >> s >> qty;
```

Но определенная здесь функция возвращает void, а не istream&. Это не слишком хорошо для отправки ей какого-либо сообщения. Рекомендуем вам переопределить возвращаемый тип в istream&.

В листинге 16.12 представлена программа, реализующая класс String и выполняющая перегрузку operator >> как "друга" класса. Возвращение ссылки на istream поддерживает цепочные операции. Вывод программы показан на рис. 16.9.

```
Enter customer name and quantity:  Simons  25
The customer name is: Simons
Quantity ordered is:  25
```

Рис. 16.9. *Вывод программы из листинга 16.12*

Листинг 16.12. Перегрузка операции ввода для типа, определенного программистом

```
#include <iostream>
using namespace std;

class String {
  int size;                                    // размер строки
  char *str;                                   // начало внутренней строки
  void set(const char* s);                     // выделение закрытой строки
public:
friend istream& operator >> (istream& in, String& s);
  String (const char* s = "")                  // по умолчанию и преобразование
    { set(s); }
  String (const String &s)                     // конструктор копирования
    { set(s.str); }
  ~String()                                    // деструктор
  { delete [ ] str; }
  String& operator = (const Strings s);        // присваивание
  char* get () const                           // возвращение указателя в начало
  { return str; }
} ;

void String::set(const char* s)
   { size = strlen(s);                         // оценка размера
     str = new char[size + 1];                 // запрос памяти динамически
                                               // распределяемой области
     if (str == 0) { cout << "Out of memory\n" exit(0); }
     strcpy(str.s); }                          // копирование клиентских данных
                                               // в динамически распределяемую область

String& String::operator = (const String& s)
{  if (this == &s) return *this;               // ничего, если самоприсваивание
   delete [ ] str;                             // возвращение существующей памяти
   set(s.str);                                 // выделение/установка новой памяти
   return *this; }                             // для поддержки цепочечного присваивания

istream& operator >> (istream& in, String& s)  // глобальный "друг"
{  char name[80];                              // локальная память для данных
   in >> name;                                 // принять данные
   delete [] s.str;                            // возвращение существующей памяти
   s.set(name);                                // выделение/установка новой памяти
   return cin; }                               // важно для цепочечных действий

int main()
{  String s; int qty;                          // локальные переменные
   cout << "Enter customer name and quantity: ";
   cin >> s >> qty;                            // принять имя, количество
   cout << "The customer name is: ";
   cout << s. get() << endl;                   // использование общедоступных методов
   cout << "Quantity ordered is: ";
   cout << qty << endl;
   return 0 ;
   }
```

Этот пример показывает, что типы, определенные программистом, должны интерпретироваться таким же образом, как и встроенные типы C++.

Перегруженная операция <<

Подобно operator >>, операция вывода operator << может быть перегружена в типы, определенные программистом.

Подобно перегруженной операции ввода operator >>, реализация операции вывода как функции-члена типа, определенного пользователем, например String, не слишком удачная мысль. Вам потребуется использовать громоздкий синтаксис, где объект String располагается слева от оператора, а объект вывода cout — справа.

```
String s;
s << cout;                  // то же, что и s.operator << (cout);
```

Идея реализации операции вывода как функции-члена библиотеки потокового класса ostream является также не очень удачной. Единственная возможность — реализовать ее как глобальную функцию. Убедитесь в том, что объект ostream является первым параметром, а не вторым. Иначе придется столкнуться с синтаксисом, в котором объект String должен быть слева от оператора.

```
void operator << (ostream& out, const String& s)
{ out << s.get(); }
```

Эту функцию не требуется делать "дружественной" для типа, определенного программистом, поскольку у нее имеется доступ ко всей необходимой информации. Независимо от того, имеется или отсутствует у этой функции доступ к данным, большинство программистов превратило бы ее в "друга".

Данная функция хорошо подходит для вывода компонентов по отдельности, но не для цепочечных операций.

```
cout << "The customer name is: ";
cout << s;
cout << endl;
```

Это неудобно. Подобно перегруженной операции ввода, используйте возвращение ссылки на объект (здесь на объект класса вывода ostream).

```
ostream& operator << (ostream& out, const String& s)
{ return out << s.get(); }
```

Теперь становится возможным объединение в цепочку операции вывода для типов, определенных пользователем, таким же образом, как и для встроенных типов:

```
cout << "The customer name is: << s << endl;          // красивый синтаксис
```

В листинге 16.13 показана программа, которая реализует класс String и перегружает operator >> как "друга" класса. Возврат ссылки istream поддерживает цепочечные операции. Вывод программы представлен на рис. 16.10.

```
Enter customer name and quantity:  Smith  42
The customer name is: Smith
Quantity ordered is:  42
```

Рис. 16.10. *Вывод программы из листинга 16.13*

Несмотря на то, что этой же цели можно достичь, написав специализированные функции-члены для ввода и вывода данных объекта, перегруженные операции придают программам на C++ элегантный вид.

Листинг 16.13. Перегрузка операций ввода и вывода для типа, определенного программистом

```
#include <iostream>
using namespace std;

class String {
  int size;                                  // размер строки
  char *str;                                 // начало внутренней строки
  void set(const char* s);                   // выделение закрытой строки
public:
friend istream& operator >> (istream& in, String& s);
friend ostream& operator << (ostream& out, const String& s);
  String (const char* s = "")                // по умолчанию и преобразование
    { set(s); }
  String (const String &s)                   // конструктор копирования
    { set(s.str); }
  ~String()                                  // деструктор
  { delete [ ] str; }
  String& operator = (const String& s);      // присваивание
  char* get () const                         // возвращение указателя в начало
  { return str; }
) ;

void String::set(const char* s)
   { size = strlen(s);                       // оценить размер
     str = new char[size + 1];               // запрос динамически распределяемой
                                             // области памяти
     if (str == 0) { cout << "Out of memory\n"; exit(0); }
     strcpy(str,s); }                        // копирование клиентских данных в динамически
                                             // распределяемую область памяти

String& String::operator = (const String& s)
{ if (this == &s) return *this;              // ничего, если самоприсваивание
  delete [ ] str;                            // возвращение существующей памяти
  set(s.str);                                // выделение/установка новой памяти
  return *this; }                            // для поддержки цепочечного присваивания

istream& operator >> (istream& in, String& s)
{ char name[80];                             // локальная память для данных
  in >> name;                                // принять данные
  delete [] s.str;                           // возвращение существующей памяти
  s.set(name);                               // выделение/инициализация новой памяти
  return cin; }

ostream& operator << (ostream& out, const String& s)
  { return out << s.str; }                   // допускается для "друзей"

int main ()
{ cout << endl << endl;
  String s; int qty;                         // локальные данные
  cout << "Enter customer name and quantity: ";
  cin >> s >> qty;                           // принять имя и количество
  cout << "The customer name is: " << s << endl;      // очень красиво
  cout << "Quantity ordered is: " << qty << endl;
  return 0;
  }
```

Итоги

В этой главе рассматривалось несколько тем, связанных одной концепцией: сделать возможным запись функций, которые разрешают клиентской программе интерпретировать объекты, определенные программистом, подобно переменным встроенных типов.

Демонстрировались унарные операции, префиксные и постфиксные операции инкремента и декремента, которые придают программам на C++ элегантный вид. Обсуждены операции преобразования. Совместно с конструкторами преобразования они продолжают тенденцию C++ в отношении правил строгого контроля за типами в пользу большей гибкости при обработке объектов.

Кроме того, повторно рассматривались операции индексирования и вызова функции — странный вид, который не следует обычным правилам преобразования синтаксиса вызова функции в синтаксис выражения. В отличие от большинства операторов C++ они могут быть перегружены только как функции-члены, а не как глобальные функции. Эти операции не очень популярны, но в случаях, когда они используются, могут быть эффективными в клиентской программе.

Вы изучили перегруженные операции ввода/вывода. Эти операции позволяют клиентской программе смешивать объекты классов, определенных программистом, и встроенных типов. Хотя с точки зрения программной инженерии они не играют какой-либо значительной роли, их применение упрощает клиентскую программу.

Перегруженные операции ввода/вывода очень популярны. Хотелось бы надеяться, что их использование доставит вам удовольствие.

Глава 17

Шаблоны как еще одно средство проектирования

Темы данной главы

- ✔ Простой пример повторного использования структуры класса
- ✔ Синтаксис определения шаблонного класса
- ✔ Шаблонные классы с несколькими параметрами
- ✔ Связи между реализациями шаблонных классов
- ✔ Специализации шаблонов
- ✔ Шаблонные функции
- ✔ Итоги

В оставшихся двух главах этой книги рассматриваются вопросы современного программирования на C++: программирование с шаблонами и программирование с исключительными ситуациями.

Обычно контейнерные классы и алгоритмы обработки (сортировка, поиск и т. п.) проектируют для компонентов конкретного типа. Если контейнер содержит набор целых величин, его нельзя использовать для хранения, например, объектов-счетов. Если функция выполняет сортировку массива целых значений, ее нельзя применять для сортировки учетных записей товарно-материальных ценностей. Часто ее невозможно использовать даже для сортировки значений двойной длины с плавающей точкой. Шаблоны C++ позволяют программисту пренебречь этим ограничением. С шаблонами можно проектировать родовые классы и алгоритмы, а затем определять, компонент какого типа следует обрабатывать конкретным объектом-контейнером или вызовом конкретной функции.

Программирование с исключительными ситуациями используется для упрощения программы, реализующей сложную логику. Обычно алгоритмы обработки применяют в C++ операторы `if` или `switch` для отделения обработки данных от обработки ошибочных или неправильно используемых данных. Для пошаговых алгоритмов сегменты исходного текста программы для основного алгоритма и для исключительных состояний записываются в альтернативных ветвях одной исходной программы. Это затрудняет ее чтение. Исключительные ситуации в C++ позволяют программисту изолировать исключительные случаи в других частях исходной программы и упростить основную обработку, что бы ее легче было понять.

Эти возможности языка, шаблоны и исключительные ситуации имеют общие характеристики. Они сложны, и на их выполнение потребуется затратить дополнительное время.

Накладные расходы памяти и времени являются непосредственным результатом сложности этих методов программирования. Если пишутся прикладные системы реального времени при жестких ограничениях на память и время выполнения, не следует использовать шаблоны и исключительные ситуации. Если приложения предположительно будут выполняться на компьютерах с достаточным количеством памяти и быстрыми процессорами, то ограничения на память и по времени не столь важны.

Рекомендуем вам постепенно вводить в свои программы такие возможности языка. Убедитесь в том, что погоня за интересными и многообещающими возможностями языка не слишком затруднит жизнь сопровождающего программиста.

В этой главе обсуждается программирование с использованием шаблонов C++. В следующей главе рассматриваются исключительные ситуации в C++ и другие расширенные возможности языка.

Простой пример повторного использования структуры класса

Подход со строгим контролем типов в C++ позволяет компилятору выявлять ошибки программирования, когда программист использует один тип вместо другого. В языке C++ имеется большое количество исключений из этого правила. Числовые значения могут заменять друг друга. Типы, определенные программистом, могут использоваться вместо других типов, предусматривая, что имеются в наличии конструкторы преобразования и операторы преобразования. Классы, связанные наследованием, также позволяют использовать ограниченную подстановку.

Имейте в виду, что существует много ограничений на использование типизированных значений. Многие алгоритмы по существу те же самые, независимо от типа значений, которыми они оперируют. Например, для поиска указанного номера счета в массиве объектов `Account` требуется просмотреть каждый компонент массива и сравнить имя владельца с указанным именем. Подобным же образом, для поиска указанной позиции в списке материально-технических ценностей в массиве позиций требуется просмотреть каждый компонент массива и сравнить идентификатор позиции с указанным идентификатором. Это одинаковые действия, но невозможно передать массив позиций учета как параметр в функцию, реализующую поиск в массиве счетов. Советуем написать другую функцию. Она будет почти идентична функции поиска счета. Отличаться будет только операция сравнения: одна функция сравнивает указанное имя с именем владельца в объекте `Account`, а вторая сравнивает указанный `id` с `id` в объекте `item`.

Контейнерные классы — стеки, запросы, списки, деревья и т. д.— могут содержать различные виды компонентов. Часто в компонентных классах компоненты обрабатываются одинаковым способом, независимо от самих компонентов. Например, операции со стеком — внесение нового компонента в верхнюю часть стека, извлечение компонента из вершины стека и проверка стека не зависят от характера компонентов. Они выполняются одинаково, независимо от того, являются компоненты символами, счетами или записями материальных ценностей. Было бы неплохо иметь возможность спроектировать общий стек и использовать его для компонентов любого типа, которые требуются приложению. Это невозможно из-за строгого контроля типов в C++. Стек символов содержит символы и не может включать объекты счетов или позиции учета материальных ценностей.

Рассмотрим стек символов. Это широко распространенная структура данных. Она используется в компиляторах, калькуляторах, диспетчерах экрана, а также в других приложениях, где наборы элементов должны поддерживать протокол LIFO (last in, first out — последний вошел, первый вышел). В примере проверки скобок в выражении из главы 8 использовался стек, названный областью памяти.

для временного хранения, как базовая структура данных. Стек в следующем примере динамически выделяет требуемое количество символов в динамически распределяемой области памяти и поддерживает операции push(), pop() и isEmpty(). Операция pop всегда извлекает верхний символ стека, который был последним внесен в стек. Следующий символ всегда заносится на вершину стека, поэтому он будет извлекаться первым.

```
class Stack {
  char *items;                // стек символьных знаков
  int top, size;              // текущая вершина, общий размер
public:
  Stack(int);                 // конструктор преобразования
  void push(char);            // помещение в вершину стека
  char pop();                 // извлечение символа с вершины
  bool isEmpty() const;       // стек пуст?
  ~Stack();          // возвращение памяти динамически распределяемой области
} ;
```

В листинге 17.1 показана реализация стека вместе с тестовым драйвером для класса. Конструктор преобразования использует список инициализации для задания начальных значений элементов данных класса: общего размера символьного массива, запрашиваемого стеком, и текущей позиции вершины стека в массиве (индекс ячейки, в которую будет занесен следующий символ). Конструктор выделяет память в динамически распределяемой области, используя размер, запрошенный клиентской программой. Если в системе отсутствует свободная память, то выполнение прекращается.

Функция push() заносит значение своего параметра в массив динамически распределяемой области памяти. Размер массива в динамически распределяемой области памяти запрашивается клиентской программой, и она должна знать объем памяти, требуемый для его алгоритма, поэтому советуем вам в случае переполнения массива прекратить его выполнение. Однако тем самым передает слишком много обязанностей в клиентскую программу. Между тем клиентская программа должна сосредоточить внимание на своем алгоритме, а не на интерфейсе пользователя для диагностических сообщений. Рекомендуем передать ответственность за переполнение серверному классу.

Одна из альтернатив обработки переполнения массива — прекращение выполнения программы. Преимущество выполнения этого в серверном классе, а не в клиентской программе, состоит в том, что клиентская программа упрощается и не содержит обработки ошибок, связанной с деталями реализации сервера. Другой альтернативой является обработка серверных проблем (переполнения) на сервере, а не в клиентской программе. Это можно выполнить, например, выделяя дополнительную память в серверном объекте в случае переполнения массива.

Спорным вопросом является объем выделяемой дополнительной памяти. В листинге 17.1 обозначается стековый массив двойной точности текущего размера, копируется содержание существующего стека в заново выделенный массив, уничтожается существующий массив и продолжаются операции. При этом используется массив в динамически распределяемой области памяти, который имеет удвоенный размер по сравнению с предыдущим вариантом. Клиентская программа полностью изолирована от этих деталей управления памятью.

Функция pop() проста — она извлекает верхний символ из стека и корректирует индекс, который указывает на вершину стека. Для больших структур данных надо отследить положение вершины и возвратить существующую память, когда, например, половина существующего массива не используется. Для приведенного простого примера этого делать не нужно. Функция pop() может проверить, является ли стек пустым, и послать сообщение (или возвратить значение), если из стека нечего извлекать. Такой подход хотя и возможен, но излишне усложняет

обмен информацией между классом стека и его клиентами. Как же поступить клиенту, если делается попытка извлечения из пустого стека? В большинстве случаев (см., например, главу 8 и примеры в листингах 8.10–8.13) пустой стек является сигналом для завершения одного этапа обработки и начала нового. Тем не менее отсутствует необходимость вовлекать серверный класс в это решение, связанное с приложением. Клиентская программа должна вызывать метод стека `isEmpty()` перед каждым вызовом `pop()` и либо вызывать метод `pop()`, если стек не является пустым, либо делать что-то еще. В листинге 17.1 алгоритм завершается. Пустой стек означает, что надо закончить обработку.

Последние два метода, метод `isEmpty()` и деструктор `Stack`, являются тривиальными. Метод `isEmpty()` проверяет, возвратился ли индекс стека в свою исходную позицию. Деструктор возвращает память в динамически распределяемую область, выделенную объекту на время его существования.

Объекты класса `Stack` могут использоваться только для сохранения элементов указанного типа, а не для всех операций. Эти объекты не предназначены для инициализации друг друга или для присваивания одного другому. Можно выполнять подобные операции с любыми переменными C++, включая объекты `Stack`. Если кто-то использует объект `Stack` при инициализации или присваивании, это не должно поддерживаться. Следовательно, добавление конструктора копирования и оператора присваивания к классу `Stack` выходит за пределы допустимого. С другой стороны, полезно объявить их прототипы закрытыми. Например, если требуется передать объект `Stack` по значению, возникнет синтаксическая ошибка.

```
Initial data: abcdefghij
New size: 8
New size: 16
Inversed data: jihgfedcba
```

Рис. 17.1.
Вывод для программы из листинга 17.1

Для иллюстрации подобного подхода в листинге 17.1 первоначально выделяется небольшой массив для объекта `Stack`. Именно поэтому можно заметить отладочные сообщения, которые говорят об изменении размера массива. Вывод программы представлен на рис. 17.1.

Листинг 17.1. Класс `Stack`, содержащий символы

```
#include <iostream>
using namespace std;

class Stack {
  char *items;                    // стек символьных знаков
  int top, size;                  // текущая вершина, общий размер
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack(int);                     // конструктор преобразований
  void push(char);                // помещение в вершину стека
  char pop();                     // извлечение верхнего символа
  bool isEmpty() const;           // стек пуст?
  ~Stack();                       // возвращение памяти динамически распределяемой области
} ;

Stack::Stack(int sz = 100) : size(sz),top(0)
{  items = new char[sz];           // выделение памяти динамически распределяемой области
  if (items==0)
  { cout << "Out of memory\n"; exit(1); } }

void Stack::push (char c)
  { if (top < size)               // обычный случай: внесение символа
      items[top++] = c;
    else                          // восстановление после переполнения стека
      { char *p = new char[size*2];     // получение дополнительной памяти для
                                        // динамически распределяемой области
```

```
        if (p == 0)                          // проверка успешного выполнения
           { cout << "Out of memory\n"; exit(1); }
        for (int i=0; i < size; i++)         // копирование существующего стека
           p[i] = items[i];
           delete [] items;          // возвращение памяти динамически распределяемой области
           items = p;                        // присоединение новой памяти
           size *= 2;                        // корректировка размера стека
           cout << "New size: " << size << endl;
        items[top++] = c; } }                // внесение символа на вершину

char Stack::pop()
{ return items[-top]; }                      // безусловное извлечение

bool Stack::isEmpty() const                  // извлечь еще что-нибудь
{ return top ==0; }

Stack::~Stack()
{ delete [] items; }                 // возвращение памяти динамически распределяемой области

int main()
{  char data[] = "abcdefghij";               // заранее подготовленные входные данные
   Stack s(4);                               // объект Stack
   int n = sizeof(data)/sizeof(char)-1;      // счетчик входных данных
   cout << "Initial data: ";
   for (int j = 0; j < n; j++)               // вывод на печать исходных данных
   { cout << data[j] << " "; }
   cout << endl;
   for (int i = 0; i < n; i++)               // внесение данных в стек
     s.push(data[i]);
   cout << "Inversed data: ";
   while (!s.isEmpty())                      // извлечение данных, пока стек не опустеет
     cout << s. pop () << " ";
   cout << endl;
   return 0;
   }
```

Проблема с такой структурой заключается в том, что если требуется контейнер с компонентами другого типа, то проектирование необходимо повторить с самого начала. Все объекты с типом предыдущего компонента должны заменяться экземплярами компонентов другого типа. Например, если нужен стек с целыми значениями вместо символов, то спецификации стека должны выглядеть следующим образом:

```
class Stack {
  int *items;                  // стек целых символов
  int top, size;               // текущая вершина, общий размер
  Stack (const Stack&);
  operator = (const Stack&);
public:
  Stack(int);                  // конструктор преобразования
  void push(int);              // внесение в вершину стека
  int pop();                   // извлечение верхнего символа
  bool isEmpty() const;        // стек пуст?
  ~Stack();                    // возвращение памяти динамически
                               // распределяемой области
} ;
```

Для стека значений двойной длины с плавающей точкой класс необходимо снова изменить.

```
class Stack {
  double *items;              // стек символов двойной длины
  int top, size;              // текущая вершина, общий размер
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack(int);                 // конструктор преобразования
  void push(double);          // внесение в вершину стека
  double pop();               // извлечение верхнего символа
  bool isEmpty() const;       // стек пуст?
  ~Stack();                   // возвращение памяти динамически
                              // распределяемой области
} ;
```

Обратите внимание, что для выполнения данной задачи недостаточно использовать обычный редактор. Чтобы получить эту структуру, необходимо изменить тип int на double при определении указателя в списке параметров push() и в pop() для типа возвращаемого значения. Определение элементов данных вершины top и размера size не должно изменяться. Кроме того, тип параметра конструктора должен остаться без изменений. Тем не менее повторное использование этой структуры требует внимания.

Другой метод повторного использования контейнера состоит в проектировании его с видовым типом "параметра". Он не соответствует ни типу, определенному программистом, ни встроенному типу. Например, класс Stack можно определить так:

```
class Stack {
  Type *items;               // стек символов типа Type
  int top, size;             // текущая вершина, общий размер
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack(int);                // конструктор преобразования
  void push(Type);           // занесение в вершину стека
  Type pop();                // извлечение верхнего символа
  bool isEmpty() const;      // стек пуст?
  ~Stack();                  // возвращение памяти динамически
                             // распределяемой области
} ;
```

Эта программа не компилируется только в том случае, когда компилятору не известен тип Type. Как только он определяется — программа компилируется. Это замечательно, поскольку упрощает задачу замещения и вызывает меньше ошибок. Требуется заместить только используемые экземпляры типа Type. Более того, тип Type невозможно определить, используя typedef, например:

```
typedef char Type;           // тип эквивалентен char
```

Это определение должно рассматриваться компилятором до обработки Stack. Компилятор заменит каждый экземпляр идентификатора Type зарезервированным словом char и скомпилирует получаемый в результате класс.

При таком подходе повторное использование структуры класса больше не повреждается случайными ошибками. Все это требуется, чтобы сгенерировать вариант стека для другого типа компонентов, поскольку надо заменить зарезервированное слово char

```
Initial data: 1234567890
New size: 8
New size: 16
Inversed data: 0987654321
```

Рис. 17.2. *Вывод для программы из листинга 17.2*

в операторе typedef на имя другого типа. Отсутствует опасность случайных ошибок. В листинге 17.2 представлена версия класса Stack, где тип Type обозначает int. Вывод этой программы показан на рис. 17.2.

Листинг 17.2. Повторное использование структуры класса для Stack, содержащего целые значения

```
include <iostream>
using namespace std;

typedef int Type;                              // определение изменяемого типа

class Stack {
   Type *items;                                // стек элементов типа Type
   int top, size;                              // текущая вершина, общий размер
   Stack(const Stack&);
   operator = (const Stack&);
public:
   Stack(int);                                 // конструктор преобразований
   void push(const Type&);                     // занесение в вершину стека
   Type pop();                                 // извлечение верхнего символа
   bool isEmpty() const;                       // стек пуст?
   ~Stack();                                   // возвращение памяти в динамически
                                               // распределяемую область памяти
} ;

Stack::Stack(int sz = 100) : size(sz),top(0)
{  items = new Type[sz];                       // выделить динамически распределяемую
                                               // область памяти
   if (items==0)
   { cout << "Out of memory\n"; exit(1); } }

void Stack::push (const Type& c)               // передача по ссылке
   { if (top < size)                           // обычный случай: занесение символа
        items[top++] = c;
     else                                      // восстановление после переполнения стека
       {  Type *p = new Type[size*2];          // получить дополнительную память в динамически
                                               // распределяемой области памяти
          if (p == 0)                          // проверка на успешность
             { cout << "Out of memory\n"; exit(1); }      // копирование существующего стека
          for (int i=0; i < size; i++)
             p[i] = items[i];
          delete [] items;                     // возвращение памяти динамически
                                               // распределяемой области
          items = p;                           // присоединение новой памяти
          size *= 2;                           // корректировка размера стека
          cout <<"New size: " << size << endl;
          items[top++] = c; } }                // занесение символа в вершину

Type Stack::pop()
{ return items[-top]; }                        // безусловное извлечение

bool Stack::isEmpty() const                    // что-либо извлечь?
{ return top == 0; }

Stack::~Stack()
{ delete [] items; }                           // возвращение памяти в динамически
                                               // распределяемую область
```

```
int main()
{  Type data[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 } ;
   Stack s(4);                          // объект-стек
   int n = sizeof(data)/sizeof(Type);   // счетчик введенных данных
   cout << "Initial data: ";
   for (int j = 0; j < n; j++)
     { cout << data[j] << " "; }        // вывод введенных данных
   cout << endl;
   for (int i = 0; i < n; i++)
     { s.push(data[i]); }               // внесение данных в стек
   cout << " Inversed data: ";
   while (!s.isEmpty())                 // извлечение, пока стек не опустеет
     cout << s.pop() << " ";
   cout << endl;
   return 0;
   }
```

Подход с использованием typedef позволяет повторно использовать структуру класса не только для встроенных типов, но также и для произвольных типов, определенных программистом,— счетов, инвентаризационных записей, прямоугольников и т. д. Здесь требуется пояснить способность типа компонента поддерживать операции, которые выполняются с объектами в контейнерном классе. Это не слишком сложно, но следует убедиться, что в программе контейнера эти операции можно распознать, часто они являются неявными.

В примере со Stack контейнерный класс создает массив компонента в динамически распределяемой области памяти. Значит, компонентный класс должен обеспечить конструктор по умолчанию. Это не проблема для встроенных типов, но может оказаться трудным для типа, определенного программистом.

Когда объект компонента вносится в контейнер с применением метода push(), используется присваивание. Если компонентный класс не обрабатывает свою память динамически, это не проблема. Если он не обрабатывает динамически выделенную область памяти, то компонентный класс должен обеспечивать перегруженный оператор присваивания. Обратите внимание, что оператор присваивания для контейнера остается закрытым.

Другим вопросом, связанным с повторным использованием, который требует поддержки со стороны компонентного класса, является передача параметра и возвращение значения из методов контейнера. Для встроенных типов данных такой вопрос будет тривиальным. Именно поэтому в листинге 17.1 метод push() получил параметр-значение, а метод pop() возвратил значение. В листинге 17.2 метод push() передает параметр как ссылку на константу, чтобы избежать проблем, связанных со снижением производительности (см. главу 11). Метод pop() все еще возвращает значение для совместимости с первой версией программы в листинге 17.1. Однако многие проектировщики контейнера избегают возвращения значений из методов контейнера и передают параметры (не константы) по ссылке.

Такой подход позволяет повторно использовать структуру контейнера в другой программе для любого типа, который поддерживает присваивание и копирование. Однако если в одной и той же программе используются стеки разных типов, этот подход не работает. Тип Type может иметь при компиляции только одно значение.

Когда тот же контейнер должен использоваться для компонентов разных типов в одной программе, следует возвратиться к способу ручного редактирования структуры. Каждый стек должен иметь другое имя, например charStack, intStack, pointStack и т. д. Советуем также отредактировать их код и интерфейсы.

```
class doubleStack {
  double *items;              // редактирование компонента
  int top, size;              // оставить тот же тип
```

```
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack (int);                  // оставить тот же тип
  void push(double);            // отредактировать тип параметра
  double pop();                 // отредактировать возвращаемый тип
  bool isEmpty() const;
  ~Stack();
} ;
```

Если исходный код для каждого класса редактируется по отдельности, то распространение будущих модификаций становится громоздким и вызывает ошибки. Уникальные имена классов засоряют пространство имен проекта и приводят к конфликтам имен.

Использование возможностей макроопределений может автоматизировать формирование новых имен классов и программ, но этот метод повторного использования способствует появлению ошибок. Есть сомнения, что в настоящее время программисты на C++ должны изучать, как писать макросы,— это устаревший подход к повторному использованию структур. Покажем, как выглядят макроопределения для этого стека, чтобы удовлетворить любопытство читателя.

```
#define MakeName(a,b) a/**/b

#define DefineStack(Type)      \
class MakeName( Type, Stack) {   \
  Type *items;                    \
  int top, size;                  \
Stack(const Stack&);              \
  operator = (const Stack&) ;     \
public:                           \
  Stack(int sz = 100) : size(sz),top(0)   \
  { items = new Type[sz];                  \
    if (items==0)                          \
      { cout << "Out of memory\n"; exit(1); } }     \
  void push(const Type& c)          \
  { if (top < size)                  \
      items [top++] = c;             \
    else                              \
      { Type *p = new Type [size*2]; \
        if (p == 0)                   \
          { cout << "Out of memory\n"; exit(1); }   \
        for (int i=0; i<size; i++)    \
            p[i] = items[i];          \
        delete [] items;              \
        items = p;                    \
        size *= 2;                    \
            cout << "New size: " << size << endl;  \
            items[top++] = c; } }      \
  Type pop()                           \
  { return items[-top]; }              \
  bool isEmpty() const                  \
  { return top == 0; }                  \
  ~Stack()                              \
  { delete [] items; }                  \
} ;
```

Клиент должен определить типы стека, используя имя DefineStack, заданное в начале макроопределения.

```
DefineStack(int);
```

Оно сгенерирует имя intStack как объединение типа (определенного в скобках) и имени стека (второй аргумент в макроопределении MakeName). Это также определение программы для стека целых значений. Затем клиент будет в состоянии объявить и использовать любой подходящий объект стека.

```
intStack s(4);
```

Поскольку весь код вмещается в одну строку, сгенерированную препроцессором, его сложно отладить. Лексическая подстановка, как часто случается с макроопределениями, может генерировать неверный код. Это не очень хороший способ повторного использования структуры класса.

Синтаксис определения шаблонного класса

C++ поддерживает еще один метод повторного использования структуры класса. Этот инструмент называется *шаблонный* класс. Вместо класса с фиксированным типом компонентов создается класс, где тип компонентов интерпретируется как параметр класса.

Имя этого параметра определяется программистом, например Type, T, Tp и т. д. (Как и для любого параметра, именно программист решает, какое имя ему присвоить.) Его действительное значение может быть как встроенным типом, так и определенным программистом. На этапе компиляции определения шаблона, как и для любого другого параметра, его фактическое значение не может быть известным. Когда клиентская программа присваивает значение объекту этого класса, она определяет фактический тип, который должен использоваться в классе вместо параметра класса.

Спецификация шаблонного класса

Ниже показаны спецификации для примера класса. Тип параметра обозначается именем Type, определенным программистом.

```
template <class Type>                  // Type задается при реализации
class Stack {
  Type *items;                         // будет использоваться фактический тип
  int top, size;
  Stack(const Stack&);
  operator = (const Stack&);
public:
  Stack (int);
  void push(const Type&);              // будет использоваться фактический тип
  Type pop();                          // будет использоваться фактический тип
  bool isEmpty() const;
  ~Stack();
} ;
```

Теоретически представлено расширенное понятие параметров функции. Когда записывается функция, каждому параметру присваивается имя, но оно является просто псевдонимом для имени, которое будет определено позже. Функция устанавливает все выполняемые операции для значений фактического аргумента. Однако значение фактического аргумента при написании функции неизвестно. Вы знакомитесь с ним только в момент вызова функции. Затем все появления имени

формального параметра в функции заменяются именем фактического аргумента и с этим значением аргумента выполняются вычисления.

Преимущество использования функций с параметрами состоит в том, что на момент проектирования алгоритма не требуется подтверждать значения, по которым выполняются вычисления. Вычисления могут быть записаны с любым значением, которое становится известным только в момент вызова функции. Если этот же алгоритм нужен в другом месте программы и для другого значения аргумента, совсем не обязательно в исходной программе реализовывать его снова. Данная функция может вызываться в других местах программы (без каких-либо изменений) с указанием имен фактических аргументов.

Подобным образом шаблоны указывают компилятору, как сгенерировать код при запросе с использованием конкретного типа. Когда пишется шаблон, типу присваивается имя, но оно является просто псевдонимом для имени, которое будет определено позже. Шаблон обозначает все операции, которые должны выполняться над значениями данного типа. На момент написания шаблона имя фактического аргумента неизвестно. Вы узнаете о нем только в момент создания объекта. В этом случае вычисления выполняются для значения подобного типа.

Преимущество использования шаблонных классов состоит в том, что в ходе проектирования алгоритма не требуется осуществлять связывание с типом, над которым выполняются вычисления. Они могут выполняться над любым типом (до тех пор, пока данный тип будет поддерживать эти операции). Этот тип станет известным только на момент присвоения значения объекту. Если данная структура потребуется в другом месте программы, но с применением другого типа, необязательно реализовывать ее повторно в исходной программе. Можно реализовать один и тот же шаблонный класс для различных типов в других местах программы. Реализации отличаются друг от друга только именем фактического типа. Если необходимо, объект может создаваться с тем же фактическим типом, что и раньше.

Класс проектируется только раз, а затем он используется как шаблон (здесь берется имя) для генерирования любого количества конкретных классов. Они обозначают фактические типы, которые должны использоваться вместо типа параметра, применяемого при проектировании класса. Другие элементы для шаблонных классов являются родовыми классами и параметризованными классами. Они могут использоваться с различными фактическими типами компонентов.

`Template` является зарезервированным словом C++. В определении класса после этого слова указывается непустой список параметров в угловых скобках. Эти угловые скобки (в отличие от круглых скобок в функциях C++) используются для обозначения типа параметров. Каждый параметр шаблонного класса в списке параметров является комбинацией зарезервированного слова `class` (класс) и идентификатора, определенного программистом.

Каждый параметр шаблона в угловых скобках представляет собой "заполнитель" для типа. Параметризованный класс (родовой класс) может иметь любое количество параметров типа. Несколько шаблонных параметров в списке разделяются запятыми.

```
template <class T1, class T2, class T3>        // три параметра типа
  class Triple;                                // объявление класса
```

Как и в других классах C++, зарезервированное слово `class` в списке параметров имеет иное значение, чем в других контекстах. Оно показывает, что идентификатор, расположенный за ним, является "заполнителем" фактического типа. Этот тип не обязательно должен быть классом. Он также может быть любым встроенным типом. Кроме того, допускается использование конкретных типов параметров выражений.

```
template <class Type, int size>                // тип и значение
  class Array;
```

Это подобно параметрам функций — в момент реализации класса Array значение указанного типа (в данном случае int) должно предоставляться клиентской программой.

Реализация шаблона

Создание объекта шаблонного класса называется *реализацией* (instantiation). Оно подобно созданию объекта любого класса в C++.

Когда клиентская программа приписывает значения конкретному объекту шаблонного класса, она должна предоставить для каждого шаблонного параметра аргумент фактического типа. Шаблон Stack не является классом, это всего лишь шаблон. Он не поддерживает прямое создание объектов Stack. Объект Stack без указания фактического типа при вызове функции является абсурдом. Например, push() без указания фактического значения, которое должно вноситься в стек.

Фактический тип определяется при реализации шаблона как имя типа в угловых скобках, которые добавляются к имени шаблонного класса. Имя объекта задается таким же образом, как и для неродовых классов. Параметры конструкторов используются там, где необходимо.

```
Stack<int> is(50);              // стек целых длиной 50 значений
Stack<char> cs(200);            // стек символов длиной в 200 символов
```

Реализация напоминает вызов функции, в которой формальные параметры являются "заместителями", принимающими значения фактических аргументов. В определении функции фактические аргументы указываются в скобках. В определении шаблона параметры шаблонов обозначаются в угловых скобках. В реализации объекта шаблонов, фактические типы перечисляются в угловых скобках. Если в определении шаблона содержится более одного параметра класса, то в реализации шаблона, между левой и правой угловыми скобками, должно указываться такое же количество фактических типов.

В отличие от вызова функции в реализации объекта шаблона могут использоваться только фактические значения периода компиляции. Они жестко запрограммированы в клиентской программе. C++ не реализует переменные, тип которых определяется во время выполнения. Вы не можете использовать переменную, значением которой является тип. Значения параметров по умолчанию не могут применяться для параметров типа шаблона.

Обратите внимание, что когда компилятор детально прорабатывает шаблонный класс, он не генерирует объектный код для класса. Он не может это выполнить, потому что фактический тип компонента класса неизвестен. Компилятор сохраняет во внутренних таблицах информацию о шаблоне, но не добавляет объектный код к объектному файлу, который соответствует исходному файлу с определением шаблона.

Компилятор, анализируя реализацию шаблона, например Stack<int> is(50), создает определение фактического класса (в данном случае Stack<int>), заменяя формальные параметры шаблона фактическими аргументами типа шаблона. Этот код не должен переходить в генерируемый файл объектного кода, поскольку класс может использоваться также и в другом исходном файле. Итак, компилятор генерирует дополнительный объектный файл для Stack<int>. Затем он генерирует объект класса Stack<int>.

```
template <class Type>
class Stack<int> {                  // тип передается во время реализации
  int *items;                       // используется фактический тип
  int top, size;
  Stack(const Stack&);
  operator = (const Stack&);
```

```
public:
  Stack(int);
  void push(const int&);          // используется фактический тип
  int pop();                      // применяется фактический тип
  bool isEmpty() const;
  ~Stack();
} ;
```

Обратите внимание, что в определении родового шаблона интерфейс функции push() был определен как ссылка на константу. Именно поэтому такая функция обозначается для Stack<int> как push (const int&). Это является избыточным для встроенных типов, но повышает эффективность для фактических типов, когда они представляют собой сложные классы. Однако функция pop() возвращает значение, а не ссылку, чтобы клиентская программа не зависела от времени жизни объекта-стека и его элементов.

Объект is типа Stack<int> имеет такую же природу, как и любой другой объект C++. В нем ничто не указывает на то, что это объект шаблонного класса, а не объект обычного класса. Данный объект можно использовать там же, где применяются объекты, не являющиеся шаблонами. Его можно передавать как параметр и отвечать на сообщения, определенные для родового класса.

```
If (!is.isEmpty())            // пересылка сообщения объекту
  DebugPrint(is);             // передача объекта как параметра
```

Имя шаблонного класса может применяться там же, где и имя нешаблонного класса, но должен указываться список параметров с именами фактических типов. Например, функция DebugPrint() должна определяться как функция, которая принимает аргументы конкретного типа, в данном случае Stack<int>.

```
void DebugPrint(Stack<int>& stack);       // прототип функции
```

Реализация шаблонных функций

В примере функция DebugPrint() является обычной функцией C++. Единственное различие между этой и другими функциями C++, представленными ранее, состоит в том, что ее параметр является объектом шаблонного класса. В теле данной функции параметр-стек интерпретируется как обычный объект C++ с известным классом.

Если алгоритм отладки одинаков для объектов стека разных типов, то, скорее всего, вам придется указать его в интерфейсе функции. Тип Stack<int> является специфическим для такой функции. С таким типом функция может принимать фактические аргументы конкретного типа, но не других типов стека. Интерфейс функции должен показывать, что допустим объект-стек любого типа. В C++ это поддерживается за счет понятия шаблонной (родовой) функции.

Шаблонная функция определяется так же, как и шаблонный класс. Вслед за зарезервированным словом template в угловых скобках указывается список параметров типа. Каждая запись в списке параметров состоит из зарезервированного слова class и идентификатора, который является "заместителем" имени типа. За списком параметров шаблона приводится заголовок функции с именем функции и списком параметров. В списке параметров функции используется нотация реализованных шаблонных классов. Вместо фактических типов указываются имена параметров типов из списка параметров шаблона. Вот как выглядит родовая функция DebugPrint(), которая может принимать параметр-стек любого типа.

```
template <class Type>                     // список параметров шаблона
void DebugPrint(Stack<Type>& stack);      // список параметров функции
```

Та же идея используется, когда методы общего класса реализуются за фигурными скобками области действия класса. Эти функции являются видовыми функциями — они должны работать с любым типом, если он определяется как фактический тип. (Конечно, функция может ожидать некоторых свойств от объектов фактического типа, например, способности поддерживать операции присваивания, которые зависят от реализованного алгоритма функции.)

```
void push(const Type&);          // плохо, вне фигурных скобок класса
```

Из-за прототипа функции могут возникнуть проблемы. Во-первых, он должен указывать, что эта функция принадлежит к классу Stack. Во-вторых, следует обратить внимание на то, что идентификатор Type является параметром типа, определяемым позже, а не именем типа, заданного ранее. Если это не сделать, компилятор сообщит, что Type не определен.

C++ интерпретирует функции-члены шаблонных классов как шаблонные функции. Определение (или объявление) шаблонной функции начинается с зарезервированного слова template, за которым в угловых скобках следует список параметров шаблона. Каждый компонент списка параметров шаблона включает ключевое слово class, за которым располагается идентификатор, определяющий имя параметра типа.

```
template <class Type>                  // список параметров шаблона
void Stack::push(const Type&);         // лучше, но недостаточно хорошо
```

Теперь компилятору известно, что идентификатор Type — это имя параметра типа. Он будет ожидать имя фактического типа в реализации объекта класса. Еще одна проблема связана с именем класса. Используется имя Stack, но в программе нет такого класса. Имя Stack обозначает шаблонный класс, а не класс. Для компилятора имя Stack при отсутствии какого-либо спецификатора не определено. Он должен знать типы компонентов, имеющихся в Stack.

Что надо сообщить компилятору? Каким станет тип Stack, неизвестно, поскольку на момент определения класса, он может быть любого типа. Фактический тип станет известен в момент реализации. Вы знаете лишь предположительно, каким он может быть. Он будет соответствовать типу, определенному в списке параметров функции. Тем не менее это тип, который должен быть определен в угловых скобках после имени типа.

```
template <class Type>                   // список параметров шаблона
void Stack<Type>::push(const Type&);    // это достаточно хорошо
```

При использовании шаблонных классов помните об этой концепции. Подобный прототип функции строится в соответствии с теми же правилами, что и любой другой прототип функции в C++. Он определяет типы параметров функции (в данном случае Type) и классы, к которым эти функции принадлежат (здесь Stack<Type>).

Список параметров шаблона должен повторяться для каждой функции-члена, определенной вне спецификации класса.

```
template <class Type>                       // список параметров шаблона
void Stack<Type>::push (const Type& c)      // префикс шаблона
 { if (top < size)                          // обычный случай: занесение шаблона
      items[top++] = c;
   else                                // восстановление после переполнения стека
      { Type *p = new Type[size*2];     // будет использован фактический тип
        if (p == 0)
           { cout << "Out of memory\n"; exit(1); }
        for (int i=0; i < size; i++)
            p[i] = items[i];            // копирование существующих данных
```

```
    delete [] items;
    items = p;                  // присоединение нового массива динамически
                                // распределяемой области памяти
    size *= 2;                  // корректировка размера стека
    cout << "New size: " << size << endl;
    items[top++] = c; } }       // занесение символа на вершину
```

Оператор области видимости в имени функции-члена должен определять идентификаторы формальных параметров шаблонов; имя общего класса, указанного в угловых скобках после списка параметров шаблона. Это согласуется с требованиями для указания имен общих параметров каждый раз, когда имя класса упоминается вне определения класса.

Обратите внимание, что зарезервированные слова `template` или `class` не используются в префиксе шаблона операции явного задания имени — только имена типов параметров. За счет префикса шаблона формальные параметры становятся доступными. Для существующей за ними функции не нужно указывать тип параметров.

```
template <class Type>                                  // список параметров шаблона
void Stack<Type>::push<Type> (const Type& c);          // избыточность
```

Функция с именем `push<Type>` здесь не имеет смысла, достаточно имени самой функции.

Подобным образом, при определении конструкторов и деструкторов шаблона Его аргументы объявляются только один раз в префиксе шаблона, а не в имени функции-члена, например, `Stack<Type>::Stack()`, но не `Stack<Type>::Stack<Type>()`. Второй `Stack` является именем функции-члена, а не спецификатора типа.

```
template <class Type>
Stack<Type>::Stack(int sz = 100) : size(sz),top(0)
{ items = new Type[sz];
  if (items==0)
    { cout << "Out of memory\n"; exit(1); } }
```

Это справедливо и для деструктора. То, что указывается после тильды, является именем функции-члена, а не именем класса. Следовательно, оно не должно включать параметры шаблона. Часть, которая предшествует двоеточию — оператору явного задания, является именем класса и должна включать параметры шаблона.

```
template <class Type>
Stack<Type>::~Stack()                  // специальный синтаксис деструктора
  { delete [] items; }
```

Обратите внимание, что деструктор не использует имя параметра типа в теле функции. Однако параметр типа все еще должен использоваться как в списке параметров шаблона, так и в его префиксе. Это общее правило. Зарезервированное слово `template` в списке параметров типов в определении функции должно включать все параметры типов, упомянутые в списке параметров шаблона для класса. Такое утверждение справедливо для префикса шаблона, который определяет имя класса. Указывайте все параметры шаблона, даже если не все они используются в функции.

Например, функция-член `isEmpty()` проверяет значение целого индекса. Ее тело всегда одинаково и не зависит от того, какой фактический тип используется при реализации объекта. Тем не менее определение этой функции-члена требует полного списка параметров шаблона и его полного префикса.

```
template <class T>                     // не используется в функции
bool Stack<T>::isEmpty() const         // возвращает значение типа bool
  { return top == 0; }                  // одинаковое для любого типа
```

В этом определении вместо Type для параметра типа используется идентификатор T. Имя типа является "заместителем". Можно использовать любое имя до тех пор, пока оно применяется везде, где должно использоваться. В данном примере важным является требование использовать одинаковое имя в списке параметров шаблона и в префиксе шаблона той же функции. Для другой функции может применяться имя другого параметра.

В листинге 17.3 показана реализация стека как шаблонного класса. Используются три различных имени: Type, T и Tp. Для разных функций-членов эти имена совершенно независимы. Такие функции могут быть реализованы в различных исходных файлах. С точки зрения программной инженерии — это не очень хорошая идея. Классы C++ поощряют объединение связанных друг с другом вещей. Но синтаксис языка позволяет реализовать разные функции-члены одного класса в различных исходных файлах.

В данной версии программы клиентская программа реализует стек объектов Point. Объекты класса Point содержат два поля целого типа для своих координат: пустой конструктор по умолчанию (для поддержки создания массива) и простой конструктор копирования (для поддержки возвращения по значению из функции-члена, pop()). Для такого простого класса ни один из этих конструкторов не нужен. Для классов, обрабатывающих динамически распределяемую область памяти, требуются оба поля.

```
Initial data: (1,2) (3,4) (5,6) (7,8) (9,0)
New size: 8
Inversed data: (9,0) (7,8) (5,6) (3,4) (1,2)
```

Рис. 17.3. *Вывод для программы из листинга 17.3*

Для совместимости с предыдущими примерами координаты Point отображаются на экране операции operator <<. Данная операция перегружается как глобальный "друг" класса Point. Вывод программы приведен на рис. 17.3.

Листинг 17.3. Повторное использование структуры класса для Stack, содержащего объекты Point

```
#include <iostream>
using namespace std;

class Point {
       int x, y;
friend ostream& operator << (ostream& out, const Points p);
public:
      Point () { }                    // конструктор по умолчанию: пустой
      Point (const Point &p)          // конструктор копирования: для return
      { x = p.x; y = p. y; }
      void set (int a, int b)         // установить координаты Point
      { x = a; y = b; }
} ;

ostream& operator << (ostream& out, const Point& p)
{  out << "(" << p.x << "," << p.y << ")";
   return out ; }

template <class Type>
class Stack {
   Type *items;
   int top, size;                     // стек элементов типа Type
   Stack (const Stack&);              // текущая вершина, общий размер
   operator = (const Stack&);
public:
   Stack (int);                       // конструктор преобразования
   void push (const Type&);           // занесение в вершину стека
   Type pop( ) ;                      // извлечение верхнего символа
```

```
   bool isEmpty() const;                  // стек пуст?
   ~Stack ();                             // возвращение памяти динамически распределяемой области
} ;

template <class Type>
Stack<Type>: : Stack (int sz = 100) : size(sz),top(0)
{  items = new Type[sz];                  // выделение динамически распределяемой области памяти
   if (items==0)
   { cout << "Out of memory\n"; exit(1); } }

template <class T>
void Stack<T>::push (const T& c)
   { if (top < size)                      // обычный случай: занесение символа
        items[top++] = c;
     else                                 // восстановление после переполнения стека
       { T *p = new T[size*2];            // получить дополнительную память в динамически
                                          // распределяемой области
         if (p == 0)                      // проверка на успех
            { cout << "Out of memory\n"; exit(1); }
         for (int i=0; i < size; i++)     // копировать существующий стек
             p[i] = items [i];
         delete [] items;                 // возвращение памяти динамически распределяемой области
         items = p;                       // присоединение новой памяти
         size *= 2;                       // корректировка размера стека
         cout << "New size: " << size << endl;
         items [top++] = c; } }           // занесение символа в вершину

template <class Type>
Type Stack<Type>::pop()
{ return items[-top]; }                   // безусловное извлечение

template <class Tp>
bool Stack<Tp>::isEmpty() const           // что-либо извлечь
{ return top == 0; }

template <class Type>
Stack<Type>::~Stack ( )
{ delete [] items; }                      // возвращение памяти динамически распределяемой области

int main()
{
   Point data [5] ;
   data[0].set(1, 2); data[1] .set(3, 4); data[2].set (5, 6);
   data[3].set(7, 8); data[4].set(9, 0);
   Stack<Point> s(4);                     // объект Stack
   int n = sizeof (data)/sizeof (Point);        // количество компонентов
   cout << "Initial data: ";
   for (int j = 0; j < n; j++ )
     { cout << data[j] << " "; }          // вывод входных данных
   cout << endl;
   for (int i = 0; i < n; i++)
     { s.push(data[i]); }                 // занесение данных в стек
   cout << " Inversed data: ";
   while ( !s.isEmpty())                  // извлечение, пока стек не опустеет
     cout << s.pop() << " ";
   cout << endl;
   return 0 ;
   }
```

Имена функций-членов становятся доступными для использования в клиентской программе, как только объекту класса приписываются конкретные значения аргументов типа. Когда функция-член отправляется как сообщение объекту конкретного класса, имя функции в клиентской программе определяется без префиксов.

```
Stack<Point> s(4);                    // объекту присвоено конкретное значение
. . .
for (int i = 0; i < n; i++)
    { s.push(data[i]); }              // спецификаторы типа отсутствуют
```

Это справедливо для любой клиентской программы — нет указаний, что сообщения отправляются объектам шаблонных классов, а не регулярным классам. Внутри определения класса ситуация другая. За именем класса может либо следовать, либо нет список параметров в квадратных скобках. Например, для реализации контейнера в виде связного списка класс-узел будет шаблоном, содержащим указатель на следующий узел. Следовательно, определение класса-узла должно использовать имя класса для своего собственного компонента класса.

```
template <class T>
struct Node {                         // общедоступные данные
  T item;
  Node *next;                         // поле next указывает на Node
  Node(const T&);                     // конструктор
} ;
```

Здесь имя параметра не используется в предположении, что компилятор поймет, что следующее поле соответствует определяемому классу. Более строгий подход предполагает, что класс Node не существует, если не определен тип компонента-узла.

```
template <class T>
struct Node {                         // общедоступные данные
  T item;
  Node<T> *next;                      // поле next указывает на Node<T> *next
  Node(const T&);                     // конструктор
} ;
```

С точки зрения программной инженерии второй вариант класса Node лучше первого. Однако компилятор должен скомпилировать каждую версию без затруднений.

Вспомним, что реализация каждого класса генерирует отдельный экземпляр кода объекта класса. В зависимости от версии компилятора код объекта может помещаться в отдельный объектный файл, который позже связывается с другими файлами объектного кода. Как следствие этого — исходные файлы и объектные файлы не соответствуют друг другу. Существуют объектные файлы, происхождение которых с трудом устанавливает проектировщик.

Распространение реализаций шаблонов может увеличить время компиляции и компоновки, а также размер объектной программы. Некоторые компиляторы могут предложить способы возможного уменьшения этого негативного влияния.

Когда функции-члены шаблонов встраиваются в определение класса, не требуется определять префикс шаблона и оператор явного задания с именами параметров.

```
template <class Type>
class Stack {
  Type *items;                        // стек элементов типа Type
  int top, size;                      // текущая вершина, общий размер
  Stack(const Stack&);
  operator = (const Stack&);
```

```
public:
  Stack(int sz = 100) : size(sz),top(0)        // конструктор преобразования
  { items = new Type[sz];                      // выделение памяти в динамически
                                               // распределяемой области
    if (items==0)
       { cout << "Out of memory\n"; exit(1); } }

  void push(const Type& c)                     // занесение в вершину стека
  { if (top < size)                            // обычный случай: занесение символа
       items[top++] = c;
    else                                       // recover from stack overflow
       { Type *p = new Type[size*2];           // получение дополнительной памяти
                                               // в динамически распределяемой области
         if (p == 0)                           // проверка на успешность выполнения
            { cout << "Out of memory\n"; exit(1); }
         for (int i = 0; i < size; i++)        // копирование существующего стека
            p[i] = items[i];
         delete [] items;                      // возвращение памяти динамически
                                               // распределяемой области
         items = p;                            // присоединение новой памяти
         size *= 2;                            // корректировка размера стека
         cout << "New size: " << size << endl;
         items[top++] = c; } }                 // занесение символа в вершину

  Type pop()                                   // извлечение верхнего символа
  { return items[-top]; }                      // безусловное выполнение

  bool isEmpty() const                         // стек пуст?
  { return top ==0; }

  ~Stack()                                     // возвращение памяти динамически
                                               // распределяемой области
  { delete [] items; }
} ;
```

Определение этого класса выглядит почти как обычное определение класса, где тип компонента объявляется оператором typedef (см. листинг 17.2).

Вложенные шаблоны

Шаблонный класс может использовать другие шаблоны в качестве его элементов данных. Например, шаблон-стек Stack<T> с компонентами класса T может включать элемент данных шаблона типа List<T>. Компоненты списка должны быть того же самого типа, что и компоненты стека. Функции-члены шаблона-стека могут посылать сообщения объекту шаблона-списка для реализаций операций со стеком.

Пусть шаблон-список обеспечивает такие операции, как insert_as_first() и remove_first(), добавляющие компонент как первый элемент в список и удаляющие первый компонент списка.

```
template <class T>
class List {
  public:
    void insert_as_first(const T& x);
    T remove_first();
    bool empty();                          // список пуст?
  . . .} ;                                 // остальная часть класса List
```

Использование шаблона-списка как элемента данных стека упрощает реализацию стека. О реализации динамического управления памятью заботится класс List. Не требуется включать конструкторы или деструкторы или беспокоиться о переполнении памяти. Клиентская программа должна структурировать алгоритмы обработки стека так, чтобы избегать опустошения.

```
template <class T>              // тот же тип T для Stack и List
class Stack {
  List<T> lst;                  // элемент данных шаблона
public:
  void push(const T&);          // константа, ссылка на T
  T pop();                      // возвращенное значение типа T
  bool isEmpty(); } ;           // не требуется для деструктора
```

Реализация функции-члена стека становится очень простой. Методы push() вызывают соответствующую функцию-список и полагаются на управление памятью в них.

```
template <class T>
void Stack<T>::push(const T& item)
  { lst.insert_as_first(item); }          // передать работу в List
```

Функции-члены стека pop() и isEmpty() передают работу функциям-членам списка.

```
template <class T>
T Stack<T>::pop()                         // возвращение значения типа T
{ return lst.remove_first(); }            // передать работу в список

template <class T>
bool Stack<T>::isEmpty()
{ return lst.empty(); }                   // вызов подобной функции
```

Когда клиентская программа реализует объект Stack<int>, функции-члены будут следующими:

```
void Stack<int>::push(const int& item)
{ lst.insert_as_first(item); }

int Stack<int>::pop()                    // возвращение значения типа T
{ return lst.remove_first(); }
```

Когда клиентская программа объявляет экземпляр объекта стека, например Stack<int>, процесс реализации повторяется, и компилятор формирует объектный код для List<int>. В свою очередь, объект-шаблон класса List<int> может потребовать другие шаблоны.

Если объект класса List<int> не может быть реализован, реализация стека приводит к ошибке реализации. Это может быть либо из-за ошибки в шаблоне-списке, либо недопустимой операции над параметром шаблона (например, для класса не определяется сравнение, или операция класса применяется к встроенному параметру). Отладка шаблонных классов становится труднее отладки обычных классов.

Шаблонные классы с несколькими параметрами

В предыдущих примерах использовались шаблонные классы только с одним параметром типа, хотя и упоминалось, что они могут иметь несколько параметров типов.

Это могут быть параметры типов, подобные рассмотренным в предыдущих примерах, или параметры, которые имеют сходство с обычными параметрами-значениями в функциях. За счет использования параметров и совокупности параметров типов и параметров выражений повышается гибкость шаблонов C++. В то же время появляются дополнительные синтаксические сложности.

Несколько параметров типов

Рассмотрим шаблонный класс с более чем одним параметром типа. Имена таких параметров должны использоваться в рамках класса для его элементов данных, локальных переменных или параметров методов. При реализации объекта клиентская программа предоставляет имена фактических типов. В C++ используется принцип передачи параметров. Первый параметр, определенный клиентской программой, соответствует первому параметру, заданному в шаблонном классе, и т. д.

Определяя список параметров типов в шаблонном классе, следует повторить зарезервированное слово class для каждого параметра типа. Вот пример шаблонного класса для словарных статей. У класса имеются два компонента: ключевой компонент и компонент информационного наполнения. Они могут быть компонентами произвольных классов в той степени, в которой они поддерживают присваивание и конструкторы копирования.

```
template <class Key, class Data>
class DictEntry {
    Key key;                    // поле ключа
    Data info;                  // поле информации
public:
  DictEntry () { }              // пустой конструктор по умолчанию
  DictEntry(const Key& k, const Data& d)
       : key(k),info(d) {}      // инициализация компонентов данных
  Key getKey() const
  { return key; }               // возвращение значения ключа
  Data getInfo() const
  { return info; }              // возвращение значения информации
  void setKey(const Key& k)
  { key = k; }                  // установить значение ключа
  void setInfo(const Data& d)
  { info = d; }                 // установить значение информации
} ;
```

Это простая структура, поскольку набор функций get() и set() для каждого класса поля данных можно исключить, а поля данных сделать общедоступными — качество структуры останется без изменений. Но это не важно. Пример продемонстрирует использование нескольких параметров типов.

Такой класс словарных статей реализует два объекта, которые могут обрабатываться как пара. Например, объект данного класса можно возвратить из вызова функции клиентской программой. Это проще, чем передача указателя или ссылки объекту, созданному клиентской программой. Это удобнее для реализации связанных массивов или словарей, в которых объект-ключ служит индексом для поиска связанных значений информационного поля. Чтобы алгоритм поиска работал, класс Key должен поддерживать операцию сравнения дополнительно к оператору присваивания и конструктору копирования.

Отсутствует необходимость обеспечения деструктора DictEntry, поскольку сам класс не управляет динамически своей памятью (элементами данных key и info). Если любой из компонентных классов (Key или Data) обрабатывает дополнительные ресурсы и включает деструктор, он будет вызываться автоматически в процессе разрушения объекта словарной статьи. Это аналогично последней версии класса Stack, реализованного с элементом данных List. В классе List существует деструктор, но для этой версии класса Stack он не требуется.

Для реализации объекта класса DictEntry клиентская программа должна определить два фактических типа: один для поля ключа и второй для поля данных.

Можно использовать встроенные и определенные программистом типы в любом сочетании, пока они поддерживают присваивание и копирование.

```
DictEntry<Point,char*> entry;          // семантики ссылок для строк
```

В данном случае указатель символа ссылается на массив символов, который может совместно использоваться с другими объектами в программе. Как тип C++ указатель символов поддерживает присваивание и копирование. Программист клиентской части должен знать, что в этом случае используется семантика для ссылок, и избегать операций, соответствующих только семантике для значений. Кроме обеспечения совместного использования, семантика для ссылок сохраняет память и время выполнения.

Все неприятные истории, рассказанные в главе 11, связаны с тем фактом, что деструктор принудительно освобождает память динамически распределяемой области памяти. Поскольку указатель на символ как класс здесь не реализован и отсутствует деструктор, угрозы по поводу целостности программы, описанные в главе 11, к данному случаю не относятся.

В листинге 17.4 показано использование класса словарной статьи для приложения, которому требуется дать комментарии для некоторых точек на экране и найти комментарии для каждой указанной точки. Ключевым полем в словарной статье является класс Point, как и в листинге 17.3. (Здесь добавлен общий конструктор для упрощения инициализации данных и операция сравнения для облегчения поиска.) Информационное поле инициализируется указателями на строковые литералы.

После инициализации массива словарных статей main() проверяет драйвер, выполняя вывод на печать каждой записи массива. Результаты показаны на рис. 17.4.

```
Associated Data:
(1,2) Initial stage
(3,4) Analysis
(5,6) Design
(7,8) Implementation
(9,0) Testing
```

Рис. 17.4. *Вывод для программы из листинга 17.4*

Листинг 17.4. Пример шаблонного класса с двумя параметрами типа

```
#include <iostream>
using namespace std;

class Point {
      int x, y;
friend ostream& operator << (ostream& out, const Point& p);
public:
      Point() { }                              // конструктор по умолчанию: пустой
      Point(const Point &p)                    // конструктор копирования: для возвращения
      { x = p.x; y = p.y; }
      Point (int a, int b)                     // общий конструктор: установить Point
      { x = a; y = b; }
      void set(int a, int b)                   // задать координаты Point
      { x = a; y = b; }
      bool operator == (const Point& p) const
      { return x == p.x && y == p. y; }
} ;

ostream& operator << (ostream& out, const Point& p)
{  out << "(" p.x << "," << p.y << " )";
   return out ; }

template <class Key, class Data>
class DictEntry {
   Key key;
   Data info;
public:
   DictEntry () { }                            // пустой конструктор по умолчанию
```

```
  DictEntry(const Key& k, const Data& d)
        : key(k),info(d) {}                  // инициализация полей данных
  Key getKey() const
  { return key; }                            // возвращения значения key
  Data getInfo() const
  { return info; }                           // возвращения значения info
  void setKey(const Key& k)
  { key = k; }                               // задание значения key
  void setInfo(const Data& d)
  { info = d; }                              // задание значения info
} ;

int main()
{  DictEntry<Point,char*> data[5];
  data[0].setKey(Point(1,2)); data[0].setInfo("Initial stage");
  data[1].setKey(Point(3,4)); data[1].setInfo("Analysis");
  data[2].setKey(Point(5,6)); data[2].setInfo("Design");
  data[3].setKey(Point(7,8)); data[3].setInfo("Implementation");
  data[4].setKey(Point(9,0)); data[4].setInfo("Testing");
  int n = sizeof(data)/sizeof(DictEntry<Point,char*>);          // рискованно
  cout << "Associated Data:\n";
  for (int j = 0; j < n; j++)
  { cout << data[j].getKey() << " "
        << data[j].getInfo() << endl; }     // вывод на печать входных данных
  cout << endl;
  return 0;  }
```

В этом примере показано применение шаблонных классов с более чем одним параметром типа. Использование нескольких параметров вызывает вопрос о конфликтах имен. Можно ли определить несколько шаблонных классов, которые используют одинаковое имя класса? Если у класса имеется только один параметр типа, компилятор не сможет решить, какой класс использовать, когда клиентская программа создаст экземпляр объекта класса.

А как насчет нескольких параметров? Можно ли определить шаблонные классы, предусматривая, что они имеют различное количество параметров типа?

Ответ — нет. Имена шаблонов класса не могут перегружаться. Программа содержит только один шаблон класса с указанным именем.

Шаблоны с параметрами — константные выражения

Как уже упоминалось, параметры шаблона также могут быть выражениями, а не типами. Эти выражения могут быть любого встроенного или определенного программистом типа. Тип указывается явно в определении шаблона (а не как параметр типа). Клиентская программа подставляет значение данного типа во время реализации объекта.

Ниже приведен пример шаблона Stack, в котором исходный размер массива в динамически распределяемой области памяти указывается как параметр шаблона, а не как параметр конструктора.

```
template <class Type, int sz>          // параметр-выражение
class Stack {
  Type *items;                         // стек элементов типа Type
  int top, size;                       // текущая вершина, общий размер
  Stack(const Stack&);
  operator = (const Stack&);
```

```
public:
   Stack();                          // конструктор по умолчанию
   void push(const Type&);           // занесение в вершину стека
   Type pop();                       // извлечение верхнего символа
   bool isEmpty() const;             // стек пуст?
   ~Stack();                         // возвращение памяти динамически
                                     // распределяемой области
};
```

Когда создается экземпляр объекта этого класса, совсем недостаточно определить фактический тип элементов массива. Необходимо указать размер массива.

```
Stack<int,4> s;                      // объект s стека
```

Обратите внимание, что объект-стек сам не содержит каких-либо параметров. Конструктору по умолчанию они не требуются. Заметьте, что параметр должен передаваться по значению. Информация пересылается только в одном направлении, от клиентской программы к объекту шаблона. Параметры-ссылки не допускаются. Для этой цели могут использоваться только обычные параметры функции.

Параметры — шаблонные выражения, по своей сути константы, хотя в определении параметра не используется модификатор const. Они не могут изменяться в коде шаблонных функций. Их фактические значения могут быть только константными выражениями. Не допускается использование переменной, не являющейся константой, как фактического аргумента для реализации шаблона. При реализации в качестве фактических параметров разрешаются только литеральные значения или идентификаторы, помеченные как const.

```
int size = 4; const int length = 4;
Stack<int,size> s;                   // синтаксическая ошибка: не константа
Stack<int,length> s;                 // OK: константа во время компиляции
```

Когда функции-члены шаблона реализуются за пределами определения класса, в списке должны быть перечислены все параметры шаблона. Если класс-шаблон содержит параметры-выражения, последние должны быть перечислены как в списке параметров шаблонов, так и в префиксе шаблона класса.

```
template <class Type, int sz>        // параметр-выражение
Stack<Type,sz>::Stack()              // параметр-выражение
  : size(sz),top(0)
{ items = new Type[sz];              // использование параметра-выражения
  if (items==0)
  { cout << "Out of memory\n"; exit(1); } }
```

Подобно параметрам типа, параметры-выражения являются "заместителями". Их имена не важны до тех пор, пока они не противоречат функции. Они не должны противоречить различным функциям шаблонного класса. Приведем еще одну функцию-стек, в которой используются различные имена для параметра типа и для параметра-выражения.

```
template <class T, int s>                // разные имена для параметров
void Stack<T,s>::push (const T& c)       // непротиворечивые имена параметров
  { if (top < size)                      // обычный случай: занесение символа
       items [top++] = c;
    else
       { T *p = new T[size*2];           // получить дополнительную память
                                         // в динамически распределяемой области
         if (p == 0)
            { cout << "Out of memory\n"; exit(1); }
```

```
for (int i=0; i < size; i++)      // копирование существующего стека
    p[i] = items[i];
delete [] items;                  // возвращение памяти динамически
                                  // распределяемой области
items = p;
size *= 2;                        // корректировка размера стека
cout << "New size: " << size << endl;
items[top++] = c; } }             // занесение символа на вершину
```

Подобно параметрам типа, параметры выражения должны перечисляться как в списке параметров шаблона, так и в префиксе имени класса для всех функций-членов. Даже если параметр-выражение не используется в теле функции, он все равно должен быть указан в списке.

```
template <class Type, int sz>
Type Stack<Type,sz>::pop()            // параметры не используются
{ return items[-top]; }.

template <class Tp, int s>
bool Stack<Tp,s>::isEmpty() const     // параметры не используются
{ return top ==0; }

template <class Type, int sz>
Stack<Type,sz>::~Stack()
{ delete [] items; }                  // параметры не используются
```

Основная характеристика шаблонных классов с параметрами выражениями состоит в том, что каждый создаваемый экземпляр представляет другой тип C++. Как разные типы они не совместимы, и объект одного типа не может использоваться там, где ожидается объект другого типа.

```
Stack<int,4> s;                   // объект-стек
Stack<int,8> s1;                  // несовместимый объект-стек
```

Рассмотрим глобальную функцию DebugPrint(). У нее есть параметр класса Stack<int,4>. Обратите внимание, что объект параметра передается по ссылке — закрытое объявление Stack<Type,sz> конструктора копирования предотвращает передачу объектов-стеков по значению. Кроме того, заметьте, что параметр не помечен как const, поскольку он изменяется при выполнении функции.

```
void DebugPrint(Stack<int,4>& s)  // модификатор const отсутствует
{ Stack<int,4> temp;
  cout << "Debugging print: ";
  while (!s.isEmpty())            // извлечение, пока стек не опустеет
  { int x = s.pop(); temp.push(x);// сохранение во временном стеке
    cout << x << " "; }           // печать каждого компонента
  cout << endl;
  while (!temp.isEmpty())         // извлечение, пока стек не опустеет
  { s.push(temp.pop()); } }       // восстановление исходного состояния
```

Объекты стека s и s1 — это объекты разных типов. Объект s можно передать как параметр в DebugPrint(). При попытке передать подобным образом объект s1 появляется синтаксическая ошибка.

```
DebugPrint(s);                    // OK
DebugPrint(s1);                   // синтаксическая ошибка
```

Фактические выражения, определяемые одним значением, эквивалентны.

```
const int length = 4;
Stack<int,length> s2;             // совместимо со Stack<int,4>
```

Что касается только шаблонов с параметрами типа, все реализации с одинаковыми фактическими параметрами типа бывают одного типа, и один объект может использоваться вместо другого. Рассмотрим повторно шаблонный класс с одним параметром типа.

```
template <class Type>
class Stack {
  Type *itemns;                 // стек элементов типа Type
  int top, size;                // текущая вершина, общий размер
  Stack(const Stack& = 100);
  operator = (const Stack&);
public:
  Stack(int);                   // конструктор преобразования
  void push(const Type&);       // помещение на вершину стека
  Type pop();                   // извлечение верхнего символа
  bool isEmpty() const;         // стек пуст ?
  ~Stack();          // возвращение памяти динамически распределяемой области
} ;
```

Эти два объекта относятся к одному классу, поэтому один объект может использоваться вместо другого.

```
Stack<int> stackl(20);       // того же типа, что и другие объекты Stack<int>
Stack<int> stack2(50);
```

Это более гибкая и удобная реализация, чем реализация с использованием параметра-выражения. Шаблонный класс с параметрами типа и параметром-конструктором может выполнять все, что делает шаблонный класс с дополнительным параметром-выражением и без параметра конструктора. Кроме того, объекты различной исходной длины совместимы. Следует избегать шаблонов с параметрами-выражениями, за исключением тех случаев, когда очевидны их преимущества по сравнению с шаблонами с параметрами-классами.

Взаимосвязи между реализациями шаблонных классов

Реализации шаблонов могут использоваться как фактические параметры типа для реализации других шаблонных классов. Например, можно создать стек словарных статей в следующем объявлении.

```
Stack<DictEntry<Point,char*> > stackofEntries;          // 100 записей
```

Обратите внимание на большой промежуток между двумя знаками "больше". Если не включить эти пробелы, компилятор поймет код неправильно и покажет огромное количество сообщений об ошибках, совершенно не относящихся к делу. Ни одно из этих сообщений не подскажет, что требуются дополнительные пробелы. Это второй случай, когда язык C++ учитывает пробелы. Они имеют значение при определении значения параметра по умолчанию для параметра типа указателя функции, когда имя параметра не используется.

В приведенном выше объявлении реализация Stack подсказывает реализацию DictEntry. Оптимизированный компилятор может занести код объекта в кэш-память для повторного использования при выполнении компиляции. Если компилятор этого не делает, время компиляции и компоновки увеличивается.

Реализации шаблонов для различных фактических типов (и значений выражений) обособлены, и между ними нет связи или доступа друг к другу.

Например, DictEntry<int, int> и DictEntry<float, record> представляют собой два независимых индивидуальных класса. Такими же являются реализации для Stack<int> и Stack<float>. Объекты этих типов не могут заменять друг друга.

Класс может объявить все свои реализации шаблонов как имеющие общий базовый нешаблонный класс.

```
template <class T>
class Stack : public BaseStack {
. . . . } ;
```

Все реализации класса Stack будут иметь доступ к объектам BaseStack в соответствии с правилами наследования. Эти реализации не имеют доступа к необщедоступным компонентам друг друга.

Шаблонные классы как "друзья"

Нешаблонный класс (или функцию) можно объявить "другом" всех реализаций шаблонного класса, если использование созданных экземпляров объектов не зависит от их типа:

```
template <class T>
  class Stack {
    friend class StackUser;
  . . . . } ;
```

Здесь класс StackUser имеет доступ к необщедоступным компонентам любой реализации класса Stack независимо от типа, используемого в качестве фактического.

Шаблонный класс можно объявить другом нешаблонного класса, даже когда его параметр типа не связан с каким-либо фактическим значением.

```
class Node {
template <class T> friend class Stack;
  int item;
  Node *next;                          // Node *next' также OK
  Node(const int val) : item(val)
    { next = NULL; }
} ;
```

Здесь класс Node может поддерживать информационное поле и связь со следующим узлом в связанном списке. Для него не требуется никаких функций-членов. Необходим конструктор, который инициализирует оба поля данных.

Каждая реализация класса Stack является другом нешаблонного класса Node и имеет доступ к его необщедоступным компонентам. Это может быть полезным, если при вынесении за скобки общего кода уменьшается размер (и время компиляции/компоновки) объектной программы. Вместо динамически распределяемой области памяти, выделенной при реализации (или при переполнении массива), класс Stack может выделять объект Node и освобождать верхний объект Node.

Однако это не слишком полезно. Node одного типа (например, с полем информации целого типа) не сможет вместить объекты разных типов, которые клиентская программа хотела бы поместить в стек. Следовательно, класс Node также должен быть шаблоном.

Кроме того, класс Node должен определяться как шаблонный класс. Тогда объекты стека разных типов смогут создавать экземпляры и осуществлять доступ к объектам класса Node разных типов с различными типами поля item.

```
template <class Type>                  // шаблонный класс
  class Node {
          friend class Stack<T>;       // компонент любого типа
    Type item;
    Node<Type> *next;                  // Node *next' также OK
    Node(const Type& val) : item(val)
      { next = NULL; }
};
```

Технический термин для такого использования шаблонов — *неограниченные типы*. Параметр Type не зависит от параметра T и каждый параметр может принимать любые фактические значения. Каждый созданный экземпляр класса Stack (например, типа float) имеет доступ к любой реализации класса Node (например, класса Point). Данная программа не реализует в полной мере модель реального мира.

Вам следует отобразить связанные реализации так, чтобы стек целых значений стал "другом" только узла целого типа, а не узла с другими типами поля item. Чтобы получить такое отображение, нужно связать "дружественный" (клиентский) шаблонный класс (в данном случае Stack) с таким же типом, что и шаблонный класс, который предоставляет сервисы (в данном случае Node).

```
template <class Type>                          // шаблонный класс
  class Node {
           friend class Stack<Type>;           // компонент одинакового типа
    Type item;
    Node<Type> *next ;                         // Node *next', также OK
    Node(const Type& val) : item(val)
       { next = NULL; }
} ;
```

Здесь для каждого созданного экземпляра Node конкретного типа (например, класса Point) реализация Stack для того же типа (класс Point) объявляется "другом" этой реализации класса Node.

Теперь у класса Stack имеется элемент данных указателя класса Node, который инициализируется в нулевое значение с помощью конструктора Stack. Когда в стек заносится следующий узел, этот указатель ссылается на следующий узел (а новый узел указывает на узел, который обычно является первым узлом в списке). Функция-член isEmpty() проверяет, является ли данный указатель нулевым и указывает ли он на узел. Следовательно, функция pop() должна установить такой указатель в нулевое значение при удалении стека последнего узла.

```
template <class T>
class Stack {
  Node<T> *top;                 // Node *top; не допустима здесь
public:
  Stack()                       // по умолчанию: исходная длина отсутствует
  { top = NULL; }
  void push(const T&);
  T pop();
  int isEmpty() const
  { return top == NULL; }       // указывает ли вершина на узел?
  ~Stack();
} ;
```

Как и для любого шаблона, использование Node вне определения Node уточняется списком параметров. Именно поэтому элемент данных top, принадлежащий Stack, не может иметь тип Node*. Он должен быть типа Node<T>*, где T является параметром типа в Stack. В результате определения Stack создание экземпляра объекта класса Stack ведет к автоматическому заданию поля данных класса Node того же самого типа.

Метод push() класса Stack выделяет новый объект Node в динамически распределяемой области памяти. Вызов конструктора Node инициализирует поле item объекта Node в значение параметра push(). Следующее поле нового Node устанавливается в указатель на узел, на который в настоящее время указывает поле top класса Stack, а поле top устанавливается в указатель на объект нового Node.

```
template <class T>
void Stack<T>::push (const T& val)
  { Node<T> *p = new Node<T>(val);     // тип Node<T>, не Node
    if (p == NULL)
       { cout << "Out of memory\n"; exit(1); }
    p->next = top;                     // установить указатель на первый узел
    top = p; }                         // обозначает новый узел
```

В push() не надо проверять переполнение массива, потому что в этой реализации массив отсутствует. Советуем уточнить, является ли успешным выделение объекта Node. Обратите внимание на тип указателя — это Node<T>*, а не Node*. Подобным же образом, когда оператор new запрашивает память в динамически распределяемой области, тип этого оператора Node<T>, а не Node. Тип T задается при создании экземпляра Stack.

Метод pop() класса Stack устанавливает локальный указатель (типа Node<T>, а не просто Node) на первый узел стека, копирует поле информации в локальную переменную (типа T), изменяет поле top так, чтобы оно указывало на второй узел, и удаляет узел top, поскольку он больше не нужен.

```
template <class T>
T Stack<T>::pop()                  // возвращаемое значение типа T
  { Node<T> *p = top;              // Node типа T, а не Node
    T val = top->item;             // получить значение типа T
    top = top->next;               // обозначает второй узел
    delete p;                      // вернуть узел в динамически
                                   // распределяемую область памяти
    return val; }
```

Когда pop() удаляет последний узел списка (второй узел для указания в поле top отсутствует), указатель top снова становится NULL. Когда в push() заносится первый узел, оператор p->next = top устанавливает это поле в NULL. Убедитесь, что функции-члены одного класса тесно связаны друг с другом через данные класса. Исходный код функций-членов должен быть скоординирован для того, чтобы убедиться в правильном взаимодействии функции.

Деструктор Stack должен сканировать связанный список остальных узлов и возвращать их в динамически распределяемую область памяти. Снова используется локальный указатель p типа Node<T> с компонентом типа T. Он устанавливается в указатель на первый узел списка. Обратите внимание, что указатель на первый узел списка, элемент данных top, того же типа, что и указатель p: Node<T>. В цикле while указатель top перемещается на следующий узел. Узел, на который указывал указатель p, удаляется, а указатель p смещается на следующий узел.

```
template <class T>
Stack<T>::~Stack()
  { Node<T> *p = top;              // тип Node типа T
    while (top != NULL)            // если узлы отсутствуют, то top равен 0
         { top = top->next;        // указатель на следующий узел
           delete p;               // удалить предыдущий узел
           p = top; } }            // проход к следующему узлу
```

Преимущество этого подхода состоит в том, что класс Node не зависит от класса Stack. Значит, класс Node может использоваться другими "дружественными" классами, например Queue и List. Поскольку все компоненты Node (включая конструктор) являются закрытыми в этой структуре, то не "дружественные" клиенты не могут создать или осуществить доступ к объектам Node.

Другой подход состоит в обеспечении каждого клиента своим собственным серверным классом. Если определение Node вложено в закрытую часть клиента, тогда этот клиент (и его "друзья") могут осуществить доступ к классу Node.

Вложенные шаблонные классы

При использовании вложенной структуры определение шаблонного класса Node вкладывается в определение контейнерного класса, который обрабатывает объекты Node. Поскольку определение Node полностью находится в области видения класса-контейнера (например, Stack), имя Node не видно другим потенциальным клиентам (например, Queue и List). Тем не менее, компоненты Node могут стать общедоступными. Вам не нужно объявлять его единственного клиента (например, Stack) "другом" класса Node.

Это первая попытка проектирования с вложенными классами. Класс Node определяется при помощи зарезервированного слова struct, и все его компоненты являются общедоступными.

```
template <class T>
class Stack {
  template <class Type>          // Допустимо? Необходимо?
    struct Node {
       Type item;
       Node<Type> *next;         // тип зависит от реализации
       Node(const Type& val): item(val)
       { next = NULL; } } ;
  Node<T> *top;                  // элемент данных Stack
public:
  Stack()                        // по умолчанию: исходная длина не определена
  { top = NULL; }
  void push(const T&);
  T pop();
  int isEmpty() const
  { return top == NULL; }        // указывает ли top на узел?
  ~Stack();
} ;
```

При таком определении возникают проблемы. Во-первых, некоторые компиляторы не принимают определения вложенных шаблонов — они могут обрабатывать только глобальные шаблоны. Во-вторых, отсутствует необходимость использования шаблонов несвязанных типов. В этой структуре отображение классов Stack и Node относится к типу "один ко многим". Для любого типа аргумента класса Stack класс Node может использовать любой другой тип. Отображение между Stack и Node должно быть "один к одному", а не "один ко многим". Классу Stack требуется объект Node, который реализуется с тем же фактическим типом, что и сам класс Stack.

Хорошим способом достижения этого является определение класса Stack как шаблона, а затем класса Node как обычного нешаблонного класса, использующего параметр тип Stack для типов своего элемента данных и для его параметра метода.

```
template <class T>
class Stack {
    struct Node {              // это зависит от типа параметра
       T item;                 // некоторые типы как в Stack
       Node *next;             // здесь Node<T> неверно
       Node(const T& val) : item(val)      // некоторые типы как в Stack
       { next = NULL; } }
    Node *top;                 // это теперь не шаблон
public:
  Stack()                      // по умолчанию: исходная длина не определена
  { top = NULL; }
  void push(const T&);
```

```
  T pop();
  Int isEmpty() const
  { return top == NULL; }      // указывает ли top на узел?
  ~Stack();
} ;
```

Каждая реализация шаблонов Stack генерирует класс Node, который использует тот же тип, что и параметр фактического типа Stack. Тип Node в определении Stack уточнять не требуется.

Это справедливо и для функций-членов Stack. Локальный указатель в функции-члене определяется как указатель на тип Node, а не на тип Node<T>. Например, метод push() почти такой же, как и в предыдущем варианте, но указатель p определяется иначе. Для объяснения предыдущей версии указателя можно сравнить обе версии.

```
template <class T>
void Stack<T>::push (const T& val)
// { Node<T> *p = new Node<T>(val);       // тип Node<T>, а не Node
  { Node *p = new Node(val);             // тип Node, а не Node<T>
    if (p == NULL)
       { cout << "Out of memory\n"; exit(1); }
    p->next = top;                       // установить его на первый узел
    top = p; }                           // установить его на новый узел
```

```
Initial data: 1234567890
Inversed data: 0987654321
```

Рис. 17.5. *Вывод для программы из листинга 17.5*

Это же справедливо для других методов класса-контейнера. В листинге 17.5 показана реализация шаблонного класса Stack со вложенным классом Node. Вывод программы показан на рис. 17.5.

Как можно видеть, связывание типов компонентов для согласованных шаблонных классов зависит от общего подхода к проектированию. Приводит в уныние, что решение для одной структуры (например, глобальных классов) не подходит для другой структуры (например, вложенных классов). В любом случае убедитесь в том, что когда для одного класса создается экземпляр конкретного типа, то для второго класса задается экземпляр того же типа.

Листинг 17.5. Пример шаблонного класса с вложенным серверным классом

```
#include <iostream>
using namespace std;

template <class T>
class Stack {
     struct Node {                       // это зависит от типа параметра
          T item;                        // тот же тип, что и в Stack
          Node *next;                    // здесь Node<T> неверно
          Node(const T& val) : item(val) // тот же тип, что и в Stack
             { next = NULL; } } ;
  Node *top;                             // теперь это не шаблон
public:
  Stack()                                // по умолчанию: исходная длина не определена
  { top = NULL; }
  void push(const T&);
  T pop();
  int isEmpty() const
  { return top == NULL; }                // указывает ли top на узел?
  ~Stack();
} ;
```

```
template <class T>
void Stack<T>::push (const T& val)
// { Node<T> *p = new Node<T>(val);          // тип Node<T>, а не Node
   { Node *p = new Node(val);                // тип Node, а не Node<T>
     if (p == NULL)
       { cout << "Out of memory\n"; exit(1); }
     p->next = top;                          // установить его на первый узел
     top = p; }                              // установить его на новый узел

template <class T>
T Stack<T>::pop()                            // возвращаемое значение типа T
// { Node<T> *p = top;                       // тип Node<T>, а не Node
   { Node *p = top;                          // тип Node, а не Node<T>
     T val = top->item;                      // получить значение типа T
     top = top->next;                        // установить на второй узел
     delete p;                               // возвратить узел top в динамически
                                             // распределяемую область памяти
     return val; }

template <class T>
Stack<T>::~Stack()
// { Node<T> *p = top;                       // тип Node типа T
   { Node *p = top;                          // тип Node типа T
     while (top != NULL)                     // при отсутствии узлов top равен 0
          {  top = top->next;                // установить на следующий узел
             delete p;                       // удалить предыдущий узел
             p = top; } }                    // переход к следующему узлу

int main( )
{
  int data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0 } ;
  Stack<int> s;                              // объект-стек
  int n = sizeof (data)/sizeof (int);        // количество компонентов
  cout << "Initial data: ";
  for (int j = 0; j < n; j++)
    { cout << data[j] << " " ; }             // вывод на печать входных данных
  cout << endl;
  for (int i = 0; i < n; i++)
    { s. push(data[i]); }                    // занесение данных в стек
  cout << " Inversed data: ";
  while ( !s.isEmpty())                      // извлечение, пока не опустеет стек
    cout << s . pop( ) << " ";
  cout << endl;
  return 0 ;
  }
```

Шаблоны со статическими компонентами

Если в шаблонном классе объявляются статические данные, то каждая реализация шаблона будет включать отдельный набор этих статических компонентов. Все объекты, принадлежащие к конкретной реализации, совместно используют одни и те же статические компоненты. Однако они не будут иметь доступ к статическим компонентам, принадлежащим реализации параметра другого фактического типа.

Например, класс Stack может объявить свой элемент данных top статическим. Это интересная альтернатива для проектирования. Если элемент данных top объявлен статическим, то поля данных item и next могут быть перемещены в класс

Stack как элементы данных, не являющиеся статическими. Что тогда останется в классе Node? Ничего. Он становится избыточным. Следовательно, с помощью подобной структуры можно избавиться от класса Node.

В этой структуре класс Stack объединяет роли Stack из предыдущих примеров (вызовы функций-членов push(), pop() и isEmpty()) и роль класса Node (поля item и next). Именно поэтому в нем два конструктора: конструктор по умолчанию и конструктор преобразования.

Конструктор по умолчанию вызывается, когда в клиентской программе создается экземпляр объекта Stack. Он должен присутствовать, чтобы исключить синтаксическую ошибку. Конструктор преобразования вызывается из метода push(). Когда должен быть выделен новый узел, push() создает новый объект Stack, а не Node. Конструктор инициализирует поле item (в значение, которое должно сохраняться) и поле next (для указания на узел top класса Stack).

Функция pop() удаляет узел top, используя локальный указатель. Указатель имеет тип — указатель на Stack<T>. Поскольку это указатель на объект типа Stack, вызывается деструктор Stack. В предыдущих вариантах деструктор Stack удалял оставшиеся узлы стека. Здесь это опасно. Именно поэтому в классе Stack отсутствует деструктор. (Там имеется деструктор по умолчанию, который ничего не делает.)

```
template <class T> class Stack {
  static Stack *top;                          // статический элемент данных
  T item;
  Stack *next;                                // из Node
public:
  Stack() { }                                 // создать объект на клиенте
  Stack(const T& val)
        : item(val), next(top)                // создать новый узел в push()
    { top = this; }
  void push(const T& val)
    { Stack<T> *p=new Stack<T>(val); }        // нет Node<T>, нет Node
  T pop ()
    {  Stack<T> *p= top;
       T val = top->item;                     // не Node<T>, не Node
       top = top->next;                       // установить на второй узел
       delete p;                              // удалить узел top: деструктор
       return val; }
  int isEmpty() const
  { return top == NULL; }
  void remove()                               // деструктор не вызывается
  {  Stack<T> *p = top;                       // конечный указатель
     while (top != NULL)
     {  top = top->next;                      // переход к следующему узлу
        delete p;                             // удалить предыдущий вызов
        p = top; } }                          // перейти к следующему узлу
} ;
```

Отсутствие деструктора создает опасность утечки памяти. Чтобы избежать этого, класс Stack предоставляет метод remove(), выполняющий то же самое, что и деструктор класса Stack в предыдущих вариантах. Недостаток такой структуры состоит в том, что клиентская программа должна явно вызывать метод remove() для удаления оставшихся узлов в Stack.

Инициализация статического компонента шаблонного класса осуществляется не в начале выполнения программы (как для статических компонентов обычных классов), а при создании экземпляра для объекта шаблона. В этот момент создается статический компонент для этого конкретного фактического типа.

Синтаксис оператора инициализации в заголовке файла должен:

- Указывать, что статический компонент принадлежит шаблону
- Определять тип статического компонента
- Определять область видимости статического компонента
- Определять имя и исходное значение

Приведем примерный вид оператора инициализации для статического элемента данных top класса Stack. Его тип — Stack<T>*, область видимости — Stack<T>, имя — top, а исходное значение — NULL.

```
template <class T>                    // он принадлежит шаблону
  Stack<T>* Stack<T>::top = NULL;
```

Клиент может объявить только один объект указанного типа. Например, для стека целых значений создание экземпляра шаблона выглядит следующим образом.

```
Stack<int> s;                         // только один объект для типа
```

Поскольку все стеки целых величин совместно использует статический компонент, указывающий на вершину связанного списка, создание более чем одного объекта этого типа — не слишком хорошая идея.

Специализации шаблонов

В C++ понятие шаблона основывается на предположении, что алгоритм работает одинаково для различных типов данных. Поэтому имеет смысл написать только один класс. Иногда это предположение не выполняется. Алгоритм работает одинаково для различных типов данных, но для одинаковых типов некоторые детали алгоритма должны быть реализованы по-разному.

Рассмотрим шаблонный класс Array, который содержит набор данных (компонентный тип) и позволяет клиентской программе проверять, можно ли найти в наборе указанный элемент (компонентный тип).

```
template <class T>
class Array {
  T *data;                                  // массив данных в динамически
                                            // выделяемой области памяти
  int size;                                 // размер массива
  Array (const Array&);
  operator = (const Array&);
public:
  Array(T items[], int n) : size(n)         // конструктор преобразования
  { data = new T[n];                        // выделить память в динамически
                                            // распределяемой области
    if (data==0)
       { cout << "Out of memory\n"; exit(1); }
    for (int i=0; i < n; i++)               // скопировать входные данные
        data[i] = items[i]; }
  int find (const T& val) const
  { for (int i = 0; i < size; i++)
    if (val == data[i]) return i;           // вернуть индекс
    return -1; }                            // иначе вернуть -1
  ~Array ()
  { delete [] data; }
} ;
```

Этот шаблонный класс содержит только конструктор, метод find() и деструктор. Конструктор выделяет динамически распределяемую область памяти, достаточную для ввода данных, и копирует массив ввода в память динамически распределяемой области. Метод find() осуществляет поиск по памяти массива. Если значение параметра не найдено, возвращается –1. Если найдено, метод возвращает индекс для значения. Деструктор освобождает динамически распределяемую область памяти.

Клиентская программа создает экземпляр объекта Array типа int, инициализирует его и выводит на печать результаты поиска для указанного значения.

```
int main()
{ int data1 [] ={1, 2, 3, 4, 5);
  int n1 = sizeof(data1)/sizeof(int);              // число компонентов
  cout << "Initial data: ";
  for (int j = 0; j < n1; j++)
  { cout << data1[j] << " "; }                     // печать входных данных
  cout << endl;
  Array<int> a1(data1,n1);                         // объект-массив
  int item1 = 3; int idx;
  if ((idx = a1.find(item1)) != -1)
     cout << "Item " << item1 <<" is at index " << idx << endl;
  return 0; }
```

Это должно совершенно одинаково работать для целых, символов, даже для объектов Point. Для каждого из данных типов объект Array будет содержать независимую копию входных значений, и операция сравнения в методе find() будет одинаково хорошо работать. Если для объекта Array создается экземпляр для компонента типа массива символов, то конструктор и метод find() столкнутся с проблемами.

```
Array<char*> a2(data2,n2);
```

Здесь array data2[] является массивом символьных строк. Конструктор шаблона Array *скопирует указатели на строки, а не сами строки. Когда данные поступают* из жестко запрограммированного набора, не возникает проблем. В реальной жизни данные поступают из внешнего источника (а не из жестко запрограммированных массивов), и для каждого входного значения должна быть выделена независимая область памяти. Конструктор Array этого не делает. Указатели, которые он копирует в контейнер, обозначают символьный массив в памяти клиента. Подобным образом метод find() сравнивает адреса строк, а не их содержимое. Видно, что для символьного массива как компонента Array общая форма шаблонного класса не работает — требуется скопировать строки в конструктор и сравнить их в find().

C++ поддерживает концепцию *специализации* при работе с параметрами типа, которые требуют специальной обработки. Для каждого специального класса должен предусматриваться отдельный специализированный шаблон класса. Синтаксис описания специализации представляет собой объединение синтаксиса для самого шаблонного класса (в списке параметров шаблона) и инициализации шаблона в клиентской программе (в списке фактических типов). Параметр типа берется из списка параметров шаблона и перемещается в список фактических типов. Если в скобках списка параметров шаблона ничего не останется, все хорошо. Например, заголовок шаблонного класса Array:

```
template <class T>          // удалить класс T из скобок
  class Array {             // добавить <char*> к имени класса
```

становится

```
template <>                 // пустой список параметров шаблона
  class Array<char*> {      // список фактических типов
```

В методах специализации шаблонов описывается, что следует выполнить для конкретного типа. Обратите внимание: должны присутствовать как определение шаблона, так и специализированное определение шаблона. Специализация шаблона реализуется с использованием того же синтаксиса, что и для объекта шаблонного класса. Фактический тип повторяется в наименовании типа в клиентской программе.

```
Initial data: 12345
one two three four five
Item 3 is at index 2
Item three is at index 2
```

Рис. 17.6. *Вывод программы из листинга 17.6*

```
Array<char*> a1(data2,n2);    // специализированный объект шаблона
```

В листинге 17.6 показана полная программа, содержащая шаблон Array и его специализированный шаблон для компонентов типа символьного массива. Драйвер тестирования инициализирует объект шаблонного класса a1 и объект специализированного шаблона a2 и отправляет сообщения каждому объекту. Вывод программы показан на рис. 17.6.

Листинг 17.6. Пример специализации шаблонного класса

```
#include <iostream>
using namespace std;

template <class T>
class Array {
  T *data;                          // массив данных в динамически распределяемой области памяти
  int size;                         // размер массива
  Array(const Array&);
  operator = (const Array&);
public:
  Array(T items[], int n) : size(n)     // конструктор преобразования
  { data = new T[n];                // выделить память в динамически распределяемой области
    if (data==0)
      { cout << "Out of memory\n"; exit(1); }
    for (int i=0; i < n; i++)
        data[i] = items[i]; }
  int find (const T& val) const
  { for (int i = 0; i < size; i++)
        if (val == data[i])
           return i;
    return -1; }
  ~Array()
  { delete [] data; }
} ;

template <>                         // пустой список шаблона
class Array <char * > {             // тип специализации
  char* *data;                      // массив данных в динамически распределяемой области
  int size;                         // размер массива
  Array(const Array&);
  operator = (const Array&);
public:
  Array (char* items[], int n) : size(n)      // преобразование
  { data = new char* [n];           // выделить память в динамически распределяемой области
    if (data==0)
       { cout << "Out of memory\n"; exit(1); }
    for (int i=0; i < n; i++)
        { int len = strlen (items [i]);       // специально только для строк
          data[i] = new char [len+1];
          strcpy (data [i], items [i]); } }
```

```
  int find (const char*& val) const
  { for (int i = 0; i < size; i++)
       if ( strcmp (val, data [i])== 0)        // специально только для строк
            return i ;
    return -1; }
  ~Array()
  { delete [] data; }
} ;

int main()
{
  int data1[] ={1, 2, 3, 4, 5} ;
  char* data2[] = { "one", "two", "three", "four", "five" };
  int n1 = sizeof(data1)/sizeof(int);          // число компонентов
  int n2 = sizeof(data2)/sizeof(char*);
  cout << "Initial data: ";
  for (int j = 0; j < n1; j++)                 // вывод входных данных
  { cout << data1[j] << " "; }
  cout << endl ;
  for (int i = 0; i < n2; i++)
  { cout << data2[i] << " "; }                 // вывод входных данных
  cout << endl;
  Array<int> a1(data1, n1);                    // объект-массив
  Array<char*> a2(data2, n2);                  // специализированный объект
  int item1 = 3; int idx;
  char* item2 = "three";
  if ((idx = a1.find(item1)) != -1)
    cout << "Item " << item1 <<" is at index " << idx << endl;
  if ((idx = a2.find(item2)) != -1)
    cout << "Item " << item2 <<" is at index " << idx << endl;
  return 0;
  }
```

Язык C++ также поддерживает частичные специализации. Они обеспечивают специальную обработку только одного из нескольких параметров типа. Например, шаблонный класс `DictEntry` из листинга 17.4 поддерживает два типа параметров:

```
template <class Key, class Data>
class DictEntry {
  Key key;
  Data info;
public:
. . . } ;                                      // остальная часть класса
```

Для созданного экземпляра символьного массива типа `Key` специализированный шаблон создается перемещением параметра `Key` из списка параметров шаблона и добавлением специализированного типа (в угловых скобках) к имени класса.

```
template <class Data>                          // удалить тип Key
class DictEntry <char*>{                       // добавить специализированный тип
  char* key;                                   // заменить тип Key
  Data info;
public:
. . . . } ;                                    // остальная часть класса
```

На этом история не заканчивается. Вы можете выполнить специализацию любого количества параметров типа. Когда все параметры специализированы, остаются пустые угловые скобки в списке параметров шаблона. Пример для класса DictEntry:

```
template < >                          // удалить типы обоих параметров
class DictEntry <char*, char*>{       // добавить специализированные типы
  char* key;                          // заменить тип Key
  char* info;                         // заменить тип Data
public:
. . . . } ;                           // остальная часть класса
```

Если специальным образом должен интерпретироваться только второй параметр, это не проблема. Однако следует повторить первый параметр типа в списке фактических типов.

```
template <class Key>
class DictEntry <Key, char*> {
  Key key;
  char* info;
public:
. . . } ;                             // остальная часть класса
```

Когда компилятор обрабатывает созданный экземпляр шаблона, он выбирает наиболее специализированное определение, которое отвечает всем требованиям. Если подобного определения нет, компилятор использует общий шаблонный класс для создания объекта.

Использование специализации часто необходимо, когда некоторый компонентный тип требует специальной обработки. Применение специализаций усложняет программу. Не все компиляторы хорошо поддерживают специализации. Когда один из типов данных требует специальной обработки (наиболее часто это символьный массив), рассмотрите вариант написания отдельного класса с отдельным именем, например CharArray. В этом случае вы не будете сомневаться, что класс используется для создания экземпляра объекта. Однако вы не имеете гарантий в том, что подобная возможность интерпретируется одинаковым образом в различных специальных классах.

Иногда такого компромисса трудно достичь. Специализированные классы в C++ предлагают один из способов решения этой проблемы.

Шаблонные функции

Автономную функцию, не являющуюся членом класса, можно определить как шаблон. Синтаксис определения подобен синтаксису функций-членов шаблонных классов.

```
template <class T>
void swap(T& x, T& y)
{ T a = x; x = y; y = a; }
```

Когда функции требуется прототип, она также содержит список параметров шаблона, причем после каждого зарезервированного слова класса указывается параметр.

```
template <class T> void swap(T& x, T& y);
```

Определение и прототип (предварительное объявление) начинаются с зарезервированного слова template, за которым в угловых скобках указывается список

формальных параметров. Каждый формальный параметр состоит из зарезервированного слова class и идентификатора, определенного программистом. Зарезервированное слово class и идентификатор разделяются запятыми. Идентификатор в списке параметров должен указываться только один раз.

```
template <class T, class T>              // это не допускается
void swap(T& x, T& y)
{ T a = x; x = y; y = a; }
```

Каждый параметр типа должен использоваться в списке параметров шаблонной функции. Если параметр типа отсутствует в списке параметров, компилятор помечает его как синтаксическую ошибку.

```
template <class T>
  int isEmpty(void);          // ошибка компиляции для глобальной функции
```

Нешаблонные и шаблонные функции могут быть объявлены как extern, inline или static. Спецификатор располагается после списка формальных параметров шаблона и предшествует типу результата функции.

```
template <class T>
inline void swap(T& x, T& y)        // функция inline
{ T a = x; x = y; y = a; }
```

Обрабатывая определение шаблонной функции, компилятор не генерирует объектную программу. Создание экземпляра шаблонной функции осуществляется при ее вызове. Поскольку каждый фактический параметр указывается в списке параметров именем, а его тип известен компилятору, при вызове шаблона не требуется указывать тип фактического параметра.

```
int a=5, b=10; double c=3.0, d=4.0;
swap(a,b);                          // создание экземпляра для integers
swap(c,d);                          // создание экземпляра для double
```

Компилятор генерирует код swap(int&,int&) и swap(double&,doubles), поскольку знает типы фактических параметров a и b для первого вызова и c и d для второго вызова.

Возвращаемое значение не оценивается для согласования параметров. Вы можете выполнить преобразование. Однако для параметров шаблона неявные преобразования не используются. Если компилятор не может решить, какую функцию сгенерировать для точного соответствия параметров, это синтаксическая ошибка.

```
swap(a,c);          // синтаксическая ошибка: нет точного соответствия
```

Допускается перегрузка шаблонных функций. Предусматривается, что они могут различаться по типам фактических параметров или по количеству параметров.

```
template <class T>
inline void swap(T& x, T& y, T& z) // три параметра
{ T a = x; x = y; y = z; z = a; }
```

Эта функция может отличаться от функции swap() с тремя параметрами.

```
int a=5, b=10, c=20;
swap(a,b); swap(a,b,c);
```

Шаблонные функции могут специализироваться для конкретных типов. Например, символьные массивы не могут переставляться как целые, должна использоваться специализированная версия. Правила формирования специализаций функции те же, что и для специализаций шаблонных классов. Список параметров шаблона

исчерпывается, и фактические типы (в угловых скобках) распределяются между списками функции и параметров. Специализированная функция swap():

```
template < >
inline void swap <char*> (char* x, char* y)
{ char* a = new char[strlen(x)+1] ;
  char* b = new char[strlen(y)+1] ;
  if (b==NULL) { cout << "Out of memory\n"; exit(1); }
  strcpy(a,x); strcpy(b,y);                    // память должна обеспечить
                                               // вызывающая программа
  strcpy(x,b); strcpy(y,a);
  delete a; delete b; }
```

Клиентская программа:

```
char x[20]="Hello!", y[20]="Hi, there!"; int a=5, b=10;
swap(a,b);                    // создается экземпляр общей шаблонной функции
swap(x,y);                    // создается экземпляр специализированной
                              // шаблонной функции
```

Компилятор вначале осуществляет поиск нешаблонной функции. Если она обнаруживается и параметры точно совпадают, то шаблоны не принимаются во внимание. Если устанавливается соответствие более чем одного варианта нешаблонной функции, это синтаксическая ошибка.

Если не обнаруживается совпадающая нешаблонная функция, то проверяются шаблоны. Если при этом устанавливается полное соответствие и уже существует ее реализация, она используется. Новая объектная программа не генерируется. В противном случае создается экземпляр функции.

Если не обнаруживается совпадающая шаблонная функция, то нешаблонные функции проверяются с помощью неявных преобразований.

Шаблонные функции не могут вызываться или передаваться нешаблонным функциям как параметры.

Итоги

В этой главе рассматривались шаблоны C++. Если алгоритмы должны быть одинаковыми для различных типов, следует написать их всего один раз, а позже указывать, для какого фактического типа предполагается использовать алгоритм.

Однако, используя это представление на практике, вы можете столкнуться с многочисленными трудностями. Синтаксис шаблонов C++ сложен. Он усложняется также за счет использования специализаций. Иногда попытка установить, какая специализация будет вызываться, представляет собой неприятную задачу. Временами то, что работает на одном компьютере с одним компилятором, может не функционировать на другом компьютере с другим компилятором.

К тому же из-за использования шаблонов появляются дополнительные затраты памяти и снижается производительность. Именно поэтому многие программисты на C++ стараются не применять шаблоны. С другой стороны, шаблоны используются в Стандартной библиотеке шаблонов (Standard Template Library — STL). Вам следует понимать основные принципы их использования, чтобы правильно работать с библиотекой STL.

Это мощное средство. Следует использовать его с осторожностью.

Глава 18

Программирование с обработкой исключительных ситуаций

Темы данной главы

- ✔ Простые примеры обработки исключительных ситуаций
- ✔ Синтаксис исключительных ситуаций C++
- ✔ Исключительные ситуации в объектах класса
- ✔ Операции приведения типов
- ✔ Итоги

В этой главе рассматривается относительно новый вопрос для языка C++: программирование с обработкой исключительных ситуаций. Исключительные ситуации являются механизмом языка, позволяющим программисту выделить исходную программу, которая описывает исключительные ситуации, из исходной программы, описывающей основные случаи обработки. Исключительными являются ситуации, которые не должны возникать во время обычной обработки, но временами проявляются. Отделение обработки таких исключительных ситуаций от основной программы облегчает ее чтение и сопровождение.

Это определение не совсем четкое. Действительно, то, что одни программисты рассматривают как исключительную или необычную ситуацию, другие воспринимают как неотъемлемую часть операций системы. Например, когда выделяется память в динамически распределяемой области, алгоритм должен описывать, что произойдет, если запрос удовлетворяется. Возможно, программа выйдет за пределы памяти, тогда алгоритм также должен определить, что произойдет, если память недоступна. Является ли выход за пределы памяти исключительной ситуацией? Большинство программистов ответят утвердительно.

Когда программа интерактивно считывает данные от пользователя, работающего в диалоговом режиме, алгоритм определяет обработку допустимых данных. Что произойдет, если пользователь сделает ошибку и введет неверные данные? Будет создана исключительная ситуация? Большинство программистов ответят отрицательно. Ошибки в диалоговом режиме естественны, и алгоритмы обработки этих ошибок должны рассматриваться как часть функциональных возможностей основной системы, а не как что-то происходящее очень редко.

При считывании в цикле данных из файла алгоритм определяет, что происходит только при считывании следующей записи — как должны обрабатываться разные части записи. Возможно, в файле больше нет записей для считывания. Тогда в алгоритме должно определяться, что следует делать в этом случае. Является ли достижение конца файла исключительной ситуацией? Большинство программистов ответят отрицательно. Это завершение одного этапа обработки (считывание записей файла) и начало следующего (обработка данных в памяти).

Вопрос замусоривания исходной программы разнотипными вычислительными задачами достаточно важен, независимо от того, воспринимает ли программист данную ситуацию как относящуюся к основной обработке с некоторыми дополнительными исключительными случаями (первый пример), или как набор различных случаев приблизительно равной значимости (второй и третий примеры).

Чтобы вы могли принимать интеллектуальные решения по структурированию своих алгоритмов, следует знать разнообразные средства языка программирования. Посмотрим, что представляют собой исключительные ситуации (как метод программирования в C++), какой синтаксис они требуют от программиста, как их правильно применять и каких некорректных вариантов использования следует избегать.

Первоначально C++ не поддерживал обработку исключительных ситуаций и полагался на механизмы языка C, используя переменные, доступные всей программе в целом (например, `errno`), или переходы и вызовы специальных функций, имена которых зафиксированы, но содержание может определяться программистом (например, `setjmp` и `longjmp`).

Возможности C++ по обработке исключительных ситуаций являются относительно новыми в языке. Механизм исключительных ситуаций сложен. Мы имеем необходимы опыт использования исключительных ситуаций. Кроме того, при использовании исключительных ситуаций увеличивается время выполнения и размер исполняемой программы. Поэтому исключительные ситуации не рекомендуется использовать при первой предоставляющейся возможности. Однако они должны стать частью набора средств программирования.

Простой пример для обработки исключительных ситуаций

Обычно алгоритмы обработки используют операторы управления потоком, чаще всего `if` или `switch`, для отделения обычной обработки данных от обработки ошибочных или неверных данных. Для многошаговых алгоритмов часть исходной программы для основного алгоритма и для исключительных состояний записываются в разных ветвях одной и той же исходной программы. Таким образом затрудняется чтение программы — основная линия теряется среди множества исключительных и редких случаев.

Когда в функции что-то выполняется неверно, функция может не знать, что делать с ошибкой. Аварийное завершение программы может быть хорошим решением во многих ситуациях, например при попытке внести элемент в стек системы, который оказывается заполненным. С другой стороны, аварийное прекращение программы может не вызвать высвобождения таких ресурсов, удерживаемых программой, как открытые файлы или блокировки на уровне базы данных.

Другой подход состоит в установке или возвращении кода ошибки вызывающей программе для проверки и принятия действий по восстановлению, если они возможны. Например, когда клиентская программа пытается извлечь элемент из пустого стека, возвращается код ошибки. Однако это не всегда можно сделать. Если какое-либо возвращаемое значение указанного типа является допустимым для функции pop, то специальное значение, возвращаемое вызывающей программе для обозначения исключительной ситуации, может отсутствовать.

При таком подходе клиентская программа должна проверять возможные ошибки. В результате увеличивается общий размер программы, создается громоздкая клиентская программа и замедляется скорость ее выполнения. Как правило, такой подход вызывает ошибки. Некоторые функции, например конструкторы C++, не имеют возвращаемых значений. Подобный подход использоваться не может.

Определение глобальной переменной, например `errno`, для указания ошибки не работает для программ, выполняемых одновременно. Также трудно единообразно реализовать это для последовательных программ, поскольку требуется, чтобы клиентская программа тщательно проверяла значение глобальной переменной. Такие проверки засоряют клиентскую программу и затрудняют ее понимание.

С помощью таких библиотечных функций, как `setjmp` и `longjmp`, программа может передать управление действию, которое высвободит внешние источники и выполнит восстановление после ошибки. Однако в этом случае стек вернется в исходное состояние без вызова деструкторов для объектов, созданных в стеке до вызова этих функций. Следовательно, ресурсы, удерживаемые этими объектами, могли бы быть высвобождены некорректно.

Давайте рассмотрим простой пример и проанализируем вопросы, которые должны разрешить методы обработки для исключительных ситуаций. В листинге 18.1 показана программа, которая интерактивно запрашивает у пользователя значения числителя и знаменателя для дроби, вычисляет и выводит на печать значение дроби. Для вычисления результата программа использует две серверные функции, `inverse()` и `fraction()`. Первая функция возвращает обратное значение для своего аргумента. Она вызывается второй функцией `fraction()`, которая умножает свой первый параметр на значение, возвращенное `inverse()`.

Это специально придуманная структура для такой вычислительной задачи. Простая структура не позволит продемонстрировать различные опции обработки исключительных ситуаций.

Листинг 18.1. Пример программы с обработкой ошибок в клиентской программе

```
#include <iostream>
using namespace std;

inline void inverse(long value, double& answer)
{ answer = 1.0/value; }                          // answer = 1/value

inline void fraction (long numer,long denom,double& result)
{  inverse(denom, result);                       // result =1.0 / denom
   result = numer * result; }                    // result = numer/denom

int main()
{
  while (true)                                   // бесконечный цикл
  { long numer, denom; double ans;               // числитель и знаменатель
    cout << "Enter numerator and positive\n"
         << "denominator (any letter to quit): ";
    if ((cin >> numer >> denom) == 0) break;     // ввести данные
    if (denom > 0) {                             // правильный ввод
       fraction(numer,denom,ans);                // вычисление результата
       cout << "Value of the fraction: " << ans <<"\n\n";
      }
    else if (denom == 0)                         // неверный результат
       cout << "\nZero denominator is not allowed\n\n";
    else
       cout << "\nNegative denominator: " << denom <<"\n\n"; }
  return 0;
  }
```

```
Enter numerator and positive
denominator (any letter to quit): 21 42
Value of the fraction:       0.5

Enter numerator and positive
denominator (any letter to quit): 21 0

Zero denominator is not allowed

Enter numerator and positive
denominator (any letter to quit): 42 -70

Negative denominator: -70

Enter numerator and positive
denominator (any letter to quit): 42 70
Value of the fraction:       0.6

Enter numerator and positive
denominator (any letter to quit): exit
```

Рис. 18.1. *Вывод для программы из листинга 18.1*

В этой задаче нулевое значение знаменателя недопустимо и в сообщении отклоняется. Отрицательное значение делителя также не принимается. Если дробь отрицательная, необходимо сделать отрицательным числитель. Отрицательное значение делителя должно отклоняться с сообщением, в котором также выводится недопустимое значение.

Входной цикл выполняется до тех пор, пока пользователь не введет букву вместо числовых входных данных. Оператор cout возвращает нуль, а оператор break прерывает цикл. Пример вывода для программы показан на рис. 18.1.

В этом примере оба исключительных состояния (нулевой знаменатель и отрицательный знаменатель) обнаруживаются в клиентской программе, а ошибки обрабатываются сразу же по месту их обнаружения. Серверные функции inverse() и fraction() не могут столкнуться с ошибочными входными данными. Именно поэтому они выполняют вычисление результатов для вывода безоговорочно, без тестирования допустимости входных данных.

Устранение ошибки происходит здесь с помощью вывода сообщения об ошибке и повторения запроса следующих входных данных. Основная программа (вызов серверной функции fraction()) не отделяется от программы обработки ошибки, но это не вызывает в результате серьезных проблем.

Часто случается так, что ошибку можно обнаружить только после выполнения некоторой обработки в серверной программе, далеко от того места, где появилась ошибка. Некоторые из таких ошибок могут обрабатываться по месту их обнаружения. Однако для других ошибок может потребоваться дополнительная информация, отсутствующая в серверной функции, где и была обнаружена ошибка. В этом случае информация об ошибке должна возвращаться в клиентскую программу для обработки и, если возможно, для исправления. Смоделируем такую ситуацию. Передадим проверку входных данных из клиентской программы в серверную функцию inverse().

В листинге 18.2 показан этот подход к обработке ошибок. Функция inverse() вычисляет обратное значение своего аргумента. Если значение параметра равно нулю, inverse() использует константу DBL_MAX, определенную в заголовочном файле cfloat или float.h, как обратное значение. Затем она проверяет ответ, определяет допустимость результата и сообщает вызывающей программе, что произошло во время вывода.

Листинг 18.2. Пример программы с обнаружением ошибки в серверной программе

```
#include <iostream>
#include <cfloat>
using namespace std;

inline long inverse(long value, double& answer)
{  answer = (value) ? 1.0/value : DBL_MAX;
   if (answer==DBL_MAX)
      { cout << "\nZero denominator is not allowed\n\n";
        return 0; }                                    // нулевой знаменатель
   else if (value < 0)
          { return value; }                            // отрицательный знаменатель
   else
     return 1; }                                       // допустимый знаменатель
```

```
inline long fraction (long n,long d,double& result,char* &msg)
{  long ret = inverse (d, result);                  // result = 1.0 / d
   if (ret == 1)                                    // допустимый знаменатель
     { result = n * result; }                       // result = n / d
   if (ret < 0)
     msg = "\nNegative denominator: ";
   return ret; }

int main()
{
   while (true)
   { long numer, denom; double ans;                 // числитель/знаменатель
     char *msg; long ret;                           // информация об ошибке
     cout << "Enter numerator and positive\n"
          << "denominator (any letter to quit): ";
     if ((cin >> numer >> denom) == 0) break;       // ввод данных
        ret = fraction(numer,denom,ans,msg);        // вычисление ответа
     if (ret == 1)                                  // допустимый ответ
        cout << "Value of the fraction: " << ans <<"\n\n";
     else if (ret < 0)
        cout << msg << ret << "\n\n"; }              // отрицательное значение
   return 0;
   }
```

Если ответ — DBL_MAX, то функция inverse() обрабатывает ошибку с помощью вывода сообщения об ошибке или возврата нулевого значения, что указывает вызывающей программе об ошибке. Если параметр отрицательный, то функция inverse() возвращает значение, а клиент обнаруживает и обрабатывает ошибку. В ином случае inverse() возвращает 1, и это указывает вызывающей программе, что допустимо значение формального параметра answer.

```
Enter numerator and positive
denominator (any letter to quit): 42 0

Zero denominator is not allowed

Enter numerator and positive
denominator (any letter to quit): 42 -21

Negative denominator: -21

Enter numerator and positive
denominator (any letter to quit): -42 21
Value of the fraction:       -2

Enter numerator and positive
denominator (any letter to quit): exit
```

Рис. 18.2. *Вывод программы из листинга 18.2*

Функция fraction() оценивает возвращаемое значение inverse(). Если это значение 1 (допустимый результат), она вычисляет значение дроби. Если возвращенное значение отрицательное (отрицательный знаменатель), это значение передается своему собственному клиенту, и ему направляются дополнительные данные для обработки ошибки (выводится сообщение). Клиентская программа оценивает значение, возвращенное fraction(). Если это 1, то результат действительный, и главная функция отображает результат. Если возвращаемое значение fraction() отрицательное, клиентская программа выводит данное значение и сообщение, полученное от fraction(). В противном случае клиентская программа ничего не делает, поскольку ошибка (нулевой знаменатель) уже была обработана в inverse(). Результаты выполнения программы из листинга 18.2 показаны на рис. 18.2.

Разделение мест обнаружения ошибки и исправления ошибки приводит к более сложному решению. У серверных функций для использования имеются дополнительные возвращаемые значения и параметры. Прочная связь сторон вызывает сильную зависимость различных частей программы друг от друга. Клиентская программа должна подчиняться сложным соглашениям в отношении возвращаемых значений (в данном примере возвращение 1 обозначает допустимое значение параметра, возвращение нуля или отрицательного числа — неверное значение параметра) и вести себя по-разному для различных возвращаемых значений.

Это еще больше усложняет клиентскую программу и требует дополнительной документации, поэтому программисты клиентской и серверной частей успешно используют общие соглашения.

Другая проблема при этом подходе состоит в том, что серверная программа функции inverse() и fraction() вовлечена не только в обнаружение ошибки, но также в обмен сообщениями с пользователями о ее причинах. В более сложных программах важно убедиться, что каждая функция выполняет только одну функцию (каламбур). Функции, которая вычисляет обратное значение параметра, должно быть известно, как вычислить обратное значение для своего параметра, и ей не следует вникать в интерфейс пользователя. Функция, которая отвечает за интерфейс пользователя, должна знать, что сообщить пользователю, и не должна вовлекаться в другие вычисления. Эти обязанности надо разделять.

Еще одна проблема при таком подходе заключается в том, что компоненты пользовательского интерфейса распределяются по всему коду программы. Когда программа должна быть повторно укомплектована на французском, испанском, русском или другом языке, весь исходный текст программы должен быть просмотрен и изменен. Это значит напрашиваться на неприятности.

В листинге 18.3 предпринята попытка исключить возникновение последних двух недостатков. В нем приводится также дополнительный пример использования статических элементов данных и статических функций-членов. Все строки вывода передаются в класс MSG как закрытый статический массив строк. Класс предоставляет общедоступную статическую функцию, msg(), параметр которой указывает индекс используемой строки. Если индекс неверен, то вместо предполагаемой информации выдается сообщение об ошибке.

Листинг 18.3. Пример экстенсивных коммуникаций между клиентом и сервером

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
      static char* data [];                          // внутренние статические данные
public:
      static char* msg(int n)                        // общедоступный статический метод
      { if (n<1 || n > 5)                            // проверка допустимости индекса
           return data [0];
        else
           return data[n]; }                         // возвращение допустимой строки
};

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n" ,            // хранилище текстов
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction: "
} ;

inline long inverse(long value, double& answer, char* &msg)
{  answer = (value) ? 1.0/value: DBL_MAX;
  if (answer==DBL_MAX)
     { msg = MSG::msg(1) ;
       return 0; }                                   // нулевой знаменатель
  else if (value < 0)
     { msg = MSG::msg(2);
       return value; }                               // отрицательный знаменатель
```

```
  else
    return 1; }                                    // допустимый знаменатель

inline long fraction (long n,long d,double& result,char* &msg)
{ long ret = inverse(d, result,msg);               // result = 1.0 / d
  if (ret == 1)                                    // допустимый знаменатель
    { result = n * result; }                       // result = n / d
  return ret; }

int main()
{
  while (true)
  { long numer, denom; double ans;                 // числитель/знаменатель
    char *msg; long ret;                           // информация об ошибке
    cout << MSG::msg(3) << MSG::msg(4);            // запрос данных от пользователя
    if ((cin >> numer >> denom) == 0) break;       // ввод данных
       ret = fraction(numer,denom,ans,msg);        // вычисление ответа
    if (ret == 1)
       cout << MSG::msg(5) << ans <<"\n\n";        // допустимый ответ
    else if (ret == 0)
            cout << msg;                           // нулевой знаменатель
    else
       cout << msg << ret << "\n\n"; }             // отрицательное значение
  return 0;
  } ;
```

Видно, что серверные функции больше не вовлечены в пользовательский интерфейс. Программа, которая анализирует ситуацию, к сожалению, остается, но с этим почти ничего нельзя сделать. Если программа должна обнаруживать ошибку, она будет проверять некоторые соответствующие значения, в результате чего становится непонятной.

Кроме того, показано, что все компоненты пользовательского интерфейса собираются в одном месте. Это не только помогает программе привыкнуть к другим языкам, но также поддерживает пользовательский интерфейс в целом. Если приглашение для пользователя должно измениться, меняется только класс MSG. Если необходимо добавить или удалить сообщение, редактируется статический массив MSG::data[], соответственно, в методе MSG::msg() изменяется большое количество компонентов массива. Для того чтобы избежать такого изменения, число компонентов в массиве (определенное как локальное в msg()) можно вычислить как sizeof(data)/sizeof(char*). Поскольку значение числа сообщений используется только один раз, хранение его как строкового значения не опасно.

Обратите внимание на элементы, в которых используются статические данные и методы: зарезервированное слово static, инициализация данных вне границ класса, использование имени класса в операторе инициализации и в вызовах функции static, отсутствие объекта класса MSG в приложении, отсутствие конфликта имен между функциями msg::msg() и локальной переменной msg в клиентской программе.

Вывод для этого варианта программы такой же, как и для двух предыдущих версий приложения.

Понятно, что ограничение задачи функции inverse() только обнаружением ошибки и передача задачи исправления ошибки (в данном случае вывод сообщения с данными) увеличивает связь между клиентами и их серверами. В листинге 18.3 у функции inverse() имеется дополнительный параметр, передаваемый клиентом fraction() его собственному клиенту main(). В случае нулевого знаменателя достаточно сообщить только это. Такая информация передается в параметре msg

функции inverse(). В случае отрицательного знаменателя требуется сообщить значение знаменателя. inverse() воспользуется обоими параметрами msg и возвращенным ей значением для передачи информации своей вызывающей программой.

Исключительные ситуации C++ помогают обойтись без использования дополнительных параметров, возвращенных значений и сложных соглашений о вызовах.

Синтаксис исключительных ситуаций C++

Исключительные ситуации C++ позволяют программистам изменить последовательность передачи управления, когда возникает некоторое событие, например ошибка. Эти ошибки появляются во время выполнения (файл не найден, неверный индекс и т. п.). Когда C++ провоцирует исключительную ситуацию и отсутствует программа обработки, которой известно, как обработать это исключение, программа может завершиться.

Обработчики исключительных ситуаций являются частью исходного кода программы, который должен выполняться при возникновении исключительных ситуаций (например, вывод сообщения пользователю, сбор информации для анализа причин возникновения исключительной ситуации или исправление ошибки).

Организация исходного кода обработки ошибок в обработчики исключительных ситуаций может сделать передачу управления более логичной. Вместо того, чтобы выполнять все проверки в основном алгоритме и скрыть его смысл, выполнение обработки ошибок программируется в отдельной части. При таком подходе вы имеете возможность скрыть значения серверных функций, которые вовлечены в обнаружение ошибки.

Отдельная часть для исправления ошибки может находиться в том же методе, который вызвал исключительную ситуацию, в вызывающей программе данного метода, в вызывающей программе вызывающей программы и т. д. В подобном случае затрудняется проектирование с использованием исключительных ситуаций. Однако механизм исключительных ситуаций позволяет программисту передать управление для выполнения действий по исправлению организованным способом.

По-видимому, исключительные ситуации C++ позволяют программисту выделить исключительные случаи в другие части исходной программы и упростить основную обработку. За счет этого программа становится удобочитаемой. Нельзя сказать, что это всегда так. Как уже отмечалось, использование исключительных ситуаций полезно потому, что исключаются дополнительные параметры, возвращаемые значения и сложные соглашения о вызовах в функциях, которые выявляют проблемы, и функциях, которые пытаются исправить недостаток.

Когда C++ провоцирует исключительную ситуацию, может быть создан объект предопределенного класса Exception или класса, определенного программистом. Этот класс, заданный программистом, может быть получен как производный от класса Exception или может быть независимым классом. В результате проектирование с исключительными ситуациями становится более сложным для понимания.

Исключительные ситуации могут быть сгенерированы явно в операторе throw или неявно как результат недопустимой или неверной операции. Исключительные ситуации отслеживаются оператором catch, а управление передается к оператору, отследившему исключение. Оператор отслеживания (или блок операторов) исправляет ошибки. Передача управления после восстановления ошибки в блок отслеживания зависит от структуры программы. Обычно самопроизвольный возврат к месту возникновения исключительной ситуации не выполняется. Если же такое возвращение (т. е. продолжение обработки) желательно, программист должен организовать структуру программы специальным образом.

С обработкой исключительных ситуаций связаны три операции:

- Генерация исключительной ситуации
- Отслеживание исключительной ситуации
- Обозначение исключительной ситуации

Генерация исключительной ситуации означает указание на то, что обнаруживаются определенные исключительные (возможно, ошибочные) условия, которые должны быть обработаны с использованием механизма исключительных ситуаций C++, а не стандартными методами передачи управления.

Отслеживание исключительной ситуации означает обозначение части программы, которая спроектирована для реагирования на некоторое конкретное исключительное условие.

Обозначение исключительной ситуации — это указание исключительной ситуации, которая может возникнуть в пределах данного метода. Это помогает компилятору (и программистам, разрабатывающим клиентскую часть и осуществляющим сопровождение) узнать, что ожидать от функции и как она должна использоваться.

Генерация исключительной ситуации

Для генерации исключительной ситуации используется зарезервированное слово throw. Его применение показывает, что серверная программа обнаружила состояние, последовательность обработки которого ей неизвестна. Она генерирует исключительную ситуацию, надеясь, что где-нибудь (в ее клиенте или клиентах клиента) найдется часть программы, которая знает способ обработки подобной ситуации.

Зарезервированное слово throw используется в *операторе генерации* исключительной ситуации. Общий синтаксический вид оператора включает зарезервированное слово throw с операндом, который может быть значением любого типа, сгенерированным в процессе поиска программы обработки исключительной ситуации.

```
throw value;
```

Оператор throw обычно выполняется условно, после проверки каких-либо значений или связей в программе и при обнаружении, что они не отвечают требованиям. Это означает, что серверная программа выполняет оператор throw для уведомления клиента о проблемах, обнаруженных в серверной программе.

Оператор throw может содержать только один операнд любого типа. Однако некоторые компиляторы при попытке сгенерировать больше одного значения не помечают оператор throw как ошибку. Значение операнда throw используется клиентской программой, которая пытается обработать исключительную ситуацию для извлечения информации о контексте ошибки. Часто такая информация используется для определения поведения клиентской программы при исправлении ошибок.

Здесь приводится исправленный пример функции inverse(). В листинге 18.3 эта функция устанавливает возвращаемые значения или значения параметра для передачи в клиентскую программу. В этом варианте функция inverse() генерирует исключительные ситуации в двух случаях: во-первых, если обнаруживает, что знаменатель равен нулю, во-вторых, если видит, что знаменатель отрицательный.

```
inline void inverse(long value,double& answer)   // два параметра
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
       throw MSG::msg(1);                          // нулевой знаменатель
  if (value < 0)
       throw value; }                              // отрицательный знаменатель
```

Понятно, что в случае нулевого знаменателя функция генерирует значение типа символьного массива, а в случае отрицательного знаменателя — значение типа long. Не случайно это разные типы. Было бы намного труднее обработать исключительную ситуацию, если бы оба оператора throw генерировали значения одного типа. Если обе исключительные ситуации должны обрабатываться одинаково, то это не проблема. Если исключительные ситуации должны обрабатываться

по-разному, то клиентской программе потребовалось бы определять, что действительно произошло в серверной программе, которая сгенерировала исключительную ситуацию.

Сравнивая функцию inverse() с ее версией в листинге 18.1, то видите, что их интерфейсы похожи. Обе функции возвращают пустой тип и имеют только два параметра. В листинге 18.1 функция inverse() не пытается обнаружить какие-либо исключительные ситуации. Не делает это и ее клиент — fraction(). Клиентская программа main() должна обнаружить обе исключительные ситуации (нулевой знаменатель, отрицательный знаменатель) и обработать их.

В листингах 18.2 и 18.3 функции inverse() и fraction() пытались обнаружить исключительные ситуации, исправить некоторые из них (нулевой знаменатель) и дать возможность клиенту main() исправить остальные (отрицательный знаменатель). В результате получилась запутанная программа. Последний вариант inverse() генерирует обе исключительные ситуации. Он содержит некоторую часть, выполняющую анализ (чтобы решить, какие сгенерировать исключительные ситуации, если имеются), но интерфейс ее настолько же прост, как и для первого варианта в листинге 18.1. Здесь вам придется использовать дополнительную программу, которая была написана для отслеживания исключительных ситуаций.

Отслеживание исключительной ситуации

У функции inverse(), которая может сгенерировать две исключительные ситуации, есть прямой клиент — функция fraction(), вызывающая inverse(), и косвенный клиент — функция main(), вызывающая fraction(). Иерархия вызовов может быть произвольной. Если функция, в данном примере inverse(), генерирует исключительную ситуацию и не обрабатывает ее, одна из ее вызывающих программ (прямых или косвенных) должна отслеживать эту исключительную ситуацию.

Отслеживание исключительной ситуации — это процесс поиска программы, которая может обработать ошибку (программа обработки исключительных ситуаций). Для этого используется поиск по цепочке вызовов функции.

Предположим, что для отслеживания исключительной ситуации требуется зарезервированное слово catch. Действительно, в C++ имеется зарезервированное слово catch, которое используется в отслеживании исключительной ситуации. Однако этого недостаточно. Когда функция выявляет исключительные ситуации, она не может отследить их из произвольного источника исключительных ситуаций. Функция должна указать, из какой части своей программы она будет пытаться выполнить отслеживание исключительных ситуаций. Для этого надо использовать еще одно зарезервированное слово C++ — try. Оно должно сопровождаться блоком, который может сгенерировать исключительные ситуации.

Программа клиента, отвечающая за отслеживание ошибок, включает код. Он может вызвать исключительную ситуацию в *операторе try*.

```
void foo()                  // функция, отслеживающая исключительные ситуации
{ try                       // оператор try
    { statements;}          // операторы, генерирующие исключительные ситуации
  . . .}                    // остальная часть foo() с блоками отслеживания
```

Программы обработки исключительных ситуаций в C++ реализуются с использованием ключевых слов try и catch. Операторы (или вызовы метода), которые могут сгенерировать исключительные ситуации, помещаются в блоки try, а сами программы обработки исключительных ситуаций включаются в блоки catch.

За блоком try должны располагаться один или несколько *блоков catch*. Каждый блок catch содержит параметр, соответствующий исключению, которое обрабатывает этот блок.

```
void foo()                    // функция, отслеживающая исключительные ситуации
{ try
       { statements; }        // операторы, генерирующие исключительные ситуации
    catch (Type1 t1)          // блок отслеживания для генерации типа Type1
       { handler_for_Type1(); }
    catch (Type2 t2)          // блок отслеживания для генерации типа Type2
       { handler_for_Type2(); }
            . . . . .
    catch (TypeN tN)          // блок отслеживания для генерации типа TypeN
       { handler_for_TypeN(); }
  statements_executed_after_the_try_or_catch_block; }
```

Вслед за оператором try должна следовать одна структура (блок) отслеживания, обеспечивающая обработку исключительных ситуаций. Использование блока catch, которому не предшествует оператор try, является ошибкой. (Все хорошо, если имеются другие блоки catch между этим и предшествующим оператором try.) Использование оператора try, за которым отсутствует блок или блоки catch, является ошибкой.

Вспомним, что в операторе throw имеется аргумент некоторого типа — символьный массив, переменная long или даже значение некоторого типа класса, определенного программистом. Значение аргумента обычно содержит некоторую информацию о содержании ошибки. В случае функции inverse() эти данные, либо строка с выводимым сообщением, либо отрицательное значение знаменателя, должны отображаться. Если оператор throw генерирует объект типа класса, то конструктор для него должен позволять объекту передавать некоторую информацию о проблеме. Эта информация может использоваться конструктором catch для диагностики и исправления ошибки.

Если после блока try располагается несколько структур catch, то они должны иметь аргументы различных типов. Поскольку у структур catch имена отсутствуют, сигнатуры этих структур должны быть уникальными.

Если тип исключительной ситуации, сгенерированной в блоке try, "согласуется" с аргументом структуры catch, выполняется программа структуры catch и поиск останавливается. После завершения работы блока выполняются операторы, расположенные вслед за блоками catch для оператора try.

"Согласование аргумента" означает, что объект — исключительная ситуация, которая генерируется блоком try, может присваиваться параметру блока catch, означая точное соответствие, любые стандартные преобразования или любые параметры из подклассов структуры catch. Например, значение двойной длины можно отследить в блоке catch с параметром long, а объект SavingsAccount — в блоке catch с параметром Account.

После выполнения блока catch осуществляются операторы, которые следуют за блоком try, и структуры catch. При необходимости эти операторы могут содержать другие блоки try. Если оператор try не генерирует никакие исключительные ситуации, то структуры catch интерпретируются как операторы NULL. Они пропускаются.

Если исключительная ситуация была сгенерирована в середине оператора try, то выполнение оператора try завершается, находится и осуществляется структура catch и т. д. Операторы в блоке try, следующие за тем, который сгенерировал исключительную ситуацию, никогда не выполняются. Исключительная ситуация была сгенерирована потому, что эти операторы могут и не выполняться.

Что случится, если программа в блоке try генерирует исключительную ситуацию, которая не содержит структуру catch соответствующего типа? Тогда функция завершается. Блок try и программа, которая следует за структурами catch, не выполняются. Это означает, что соответствующий блок catch будет находиться в клиентской программе этой функции. Если он обнаруживается, все хорошо.

Если структура catch, которая способна обработать исключительную ситуацию, не находится даже в main(), то выполнение программы завершается.

Рассмотрим следующий вариант функции inverse(), который генерирует исключительные ситуации и пытается отследить их.

```
inline void inverse(long value,double& answer)      // два параметра
{ try                                               // начало блока try
    { if (value ==0)                                // нулевой знаменатель
        throw MSG::msg(1);
      if (value < 0)                                // отрицательный знаменатель
        throw value;
      answer = 1.0 / value; }                       // конец блока try
  catch (char* str)
    { cout << str; }                                // нулевой знаменатель
  catch (long val)
    { cout << MSG::msg(2) << val << "\n\n"; }}     // отрицательное значение
```

Если первый параметр имеет допустимое значение, блок try выполняется полностью, а блоки catch пропускаются. После блоков операторы отсутствуют, поэтому выполнение функции завершается.

Если первый аргумент равен нулю, генерируется исключительная ситуация для символьного массива и выполняется первый блок catch. Обратите внимание, что блок catch является "блоком". У него есть своя область видимости, и он ссылается на свой параметр str, а не на переменную, которая фактически была сгенерирована, MSG::msg(1).

Подобным образом, если первый параметр отрицательный, генерируется его значение и выполняется второй блок catch. Снова имя выводимого значения — val, а не value. Независимо от того, какая исключительная ситуация генерируется, оператор answer = 1.0/value никогда не выполняется. Это разумно, т. к. данный оператор должен выполняться, только если значение проходит все проверки.

Если операторы в блоке try генерируют исключительную ситуацию, в которой отсутствует блок catch для ее обработки в функции inverse(), то поиск программы обработки исключительной ситуации продолжается в fraction(), а затем в main().

В данном варианте функции inverse() операторы throw и блоки catch находятся в одной области действия функции. Синтаксически это допустимый вариант в C++. Не нужно использовать механизм обработки исключительных ситуаций, если информация об этих ситуациях не передавалась по различным функциям. В данном случае простой оператор if в inverse() дает те же самые результаты.

Другая проблема обработки исключительной ситуации связана с выполнением оставшейся части программы. После завершения выполнения функции inverse() вызывающие ее программы, fraction() и main(), не знают, появятся ли ещё какие-либо исключительные ситуации. Тем не менее, если какая-либо исключительная ситуация возникнет, то оператор, который вычисляет ответ, не выполняется, и программы, вызывающие inverse(), должны об этом знать.

Рассмотрим вариант inverse(), который генерирует исключительные ситуации, но не отслеживает их.

```
inline void inverse(long value,double& answer)  // два параметра
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
     throw MSG::msg(1);                         // нулевой знаменатель
  if (value < 0)
     throw value; }                             // отрицательный знаменатель
```

Отследим исключительные ситуации в клиентской функции fraction().

```
inline void fraction (long numer, long denom, double& result)
{ try {
    inverse (denom, result);                    // result = 1.0 / denom
    result = numer * result; }                  // result = numer / denom
  catch (char* str)
    { cout << str; }                            // нулевой знаменатель
  catch (long val)
    { cout << MSG::msg(2) << val << "\n\n"; }}  // отрицательное значение
```

Этот вариант не лучше предыдущего варианта inverse(). Исключительные ситуации должны обрабатываться в таких местах клиентской программы, где информация о них может использоваться для изменения поведения программы (в данном случае пропуска отображения результата вычислений).

В листинге 18.4 показан пример, как отмечалось ранее, искусственный, поскольку main() может обнаружить, что ввод сам по себе неверный. Система обработки исключительных ситуаций имеет смысл: inverse() обнаруживает ошибку и отправляет информацию main(), так что main() может пропустить использование неверных результатов.

В листинге 18.4 функция inverse() анализирует происходящее и генерирует две исключительные ситуации для своих вызывающих программ. Ее непосредственно вызывающая программа fraction() не содержит каких-либо программ обработки (структур catch), потому что она располагается в функции main(). Там же находится оператор, который должен быть пропущен. Поскольку fraction() не содержит каких-либо структур catch, она не имеет оператора try также и потому, что наличие оператора try без структур catch недопустимо.

Листинг 18.4. Пример генерации и отслеживания исключительных ситуаций

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
      static char* data [];                 // внутренние статические данные
public:
      static char* msg(int n)               // общедоступный статический метод
      { if (n<1 || n > 5)                   // проверка допустимости индекса
           return data[0];
        else
           return data[n]; }                // возвращение допустимой строки
};

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",    // хранилище текста
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction: "
} ;

inline void inverse(long value, double& answer)
{  answer = (value) ? 1.0/value : DBL_MAX;
   if (answer==DBL_MAX)
      throw MSG::msg(1);
   if (value < 0)
      throw value; }
```

```
inline void fraction (long numer, long denom, double& result)
{  inverse(denom, result);                          // result = 1.0 / denom
   result = numer * result; }                       // result = numer/denom

int main()
{
  while (true)
  { long numer, denom; double ans;                  // числитель/знаменатель
    cout << MSG::msg(3) << MSG:: msg(4);            // запрос данных от пользователя
    if ((cin >> numer >> denom) == 0) break;        // ввод данных
    try {
         fraction (numer, denom, ans);              // вычисление ответа
         cout << MSG::msg(5) << ans <<"\n\n";       // действительный ответ
        }
    catch (char* str)                               // нулевой знаменатель
    { cout << str; }
    catch (long val)                                // отрицательное значение
    { cout << MSG::msg(2) << val << "\n\n"; }
  }
  return 0;
}
```

Если inverse() не сгенерировала исключительные ситуации, то fraction() и main() продолжают вычисление и вывод результата и запрашивают следующий набор данных. Если inverse() генерирует исключительную ситуацию, она не обрабатывается в inverse(), потому что в ней отсутствуют соответствующие структуры catch. Поиск распространяется на fraction(). Поскольку fraction() не имеет каких-либо программ обработки исключительных ситуаций, поиск ведется в main(). Если в main() также отсутствует какие-либо программы обработки исключительных ситуаций, выполнение программы завершается.

Когда поиск распространяется до main(), здесь обнаруживается как оператор try, так и структуры catch. С точки зрения main(), источником проблемы является серверная функция fraction(). Клиента main() не заботит, получила ли fraction() исключительную ситуацию от одного из своих серверов или сгенерировала сама. Если fraction() генерирует исключительную ситуацию, выполнение блока try завершается до отображения ответа. Соответствующая программа обработки исключительной ситуации выводит сообщение, в котором используется информация, сгенерированная в inverse().

В этом примере блок try составляется из двух операторов: вызова fraction() и оператора вывода. Что произойдет, если переместить вызов fraction() за пределы блока try?

```
int main()
{ while (true)
  {  long numer, denom; double ans;                  // числитель/знаменатель
     cout << MSG::msg(3<< MSG::msg(4);          // запрос данных от пользователя
     if ((cin >> numer >> denom) == 0) break;       // ввод данных
     fraction(numer,denom,ans);                     // вычисление ответа
     try {
        cout << MSG::msg(5) << ans <<"\n\n"; }      // действительный ответ
     catch (char* str)                              // нулевой знаменатель
     { cout << str; }
     catch (long val)                               // отрицательное значение
     { cout << MSG::msg(2) << val << "\n\n"; } }    // конец цикла
     return 0; }
```

В этой структуре не сделано главное. Оператор try не будет создавать какие-либо исключительные ситуации. А блоки catch могут обработать только исключительные ситуации, происходящие из предшествующего оператора try. Когда inverse() генерирует исключительную ситуацию для fraction(), а fraction() генерирует эту исключительную ситуацию для main(), то ни один блок catch не будет обрабатывать исключительные ситуации, и выполнение программы завершится.

А что, если в блок try поместить только вызов функции, оставляя оператор вывода за пределами? Основной повод для этого состоит в том, что поскольку оператор не генерирует никакие исключительные операции, он портит драгоценное пространство в блоке try.

```
int main()
{ while (true)
  { long numer, denom; double ans;                    // числитель/знаменатель
    cout << MSG::msg(3) << MSG::msg(4);          // запрос данных от пользователя
    if ((cin >> numer >> denom) == 0) break;          // ввод данных
    try {
       fraction(numer,denom,ans); }                   // вычисление ответа
    cout << MSG::msg(5) << ans << "\n\n";             // действительный ответ
    catch (char* str)                                 // нулевой знаменатель
    { cout << str; }
    catch (long val)                                  // отрицательное значение
    { cout << MSG::msg(2) << val << "\n\n"; } }       // конец цикла
    return 0; }
```

В результате появляется синтаксическая ошибка. Оператор вывода находится между оператором try и блоками catch. Следовательно, за оператором try не следуют структуры catch. Кроме того, блоки catch не следуют непосредственно за оператором try. Что именно компилятор выдаст, можно только догадываться.

Вы можете расширить оператор try, включив в него операторы цикла. Основным поводом для этого может быть объединение различных источников исключительных ситуаций и обработка их в одной кэш-памяти структур catch.

```
int main()
{ while (true)
  { long numer, denom; double ans;                    // числитель/знаменатель
    try {
       cout << MSG::msg(3) << MSG::msg(4);       // запрос данных от пользователя
       if ((cin >> numer >> denom) == 0) break;       // ввод данных
       fraction(numer,denom,ans);                     // вычисление ответа
       cout << MSG::msg(5) << ans <<"\n\n"; }         // конец try
    catch (char* str)                                 // нулевой знаменатель
    { cout << str; }
    catch (long val)                                  // отрицательное значение
    { cout << MSG::msg(2) << val << "\n\n"; } }       // конец цикла
    return 0; }
```

Это выполнимо, если бы данная часть клиентской программы порождала бы дополнительные исключения. В целом, желательно сохранить область действия оператора try как можно более узкой, чтобы сопровождающему программисту было легче выяснить, откуда могут произойти исключительные ситуации.

Что можно сказать в отношении помещения всего цикла while в оператор try? Все будет зависеть от того, как это будет сделано. Если поместить зарезервированное слово try после открывающей скобки и оставить закрывающую скобку на своем месте, компилятору это не понравится.

```
int main()
{ try {
  while (true)
  { long numer, denom; double ans;              // числитель/знаменатель
    cout << MSG::msg(3) << MSG::msg(4);         // запрос данных от пользователя
    if ((cin >> numer >> denom) == 0) break;    // ввод данных
    fraction(numer,denom,ans);                  // вычисление ответа
    cout >> MSG::msg(5) << ans <<"\n\n"; }      // конец блока try
  catch (char* str)                             // нулевой знаменатель
  { cout << str; }
  catch (long val)                              // отрицательное значение
  { cout << MSG::msg(2) << val << "\n\n"; } }       // конец цикла
  return 0; }
```

Теперь область видимости оператора try не является вложенной в область действия цикла while. Какое бы проектное решение ни было принято, области видимости должны быть вложены корректно. В ином случае компилятор станет в тупик. Чем уже область видимости оператора try, тем лучше.

Как видно из этих примеров, проектирование с использованием программ обработки исключительных ситуаций должно ответить на три основных вопроса:

- Где сгенерировать исключительную ситуацию
- Где отследить исключительную ситуацию
- Какую информацию отправить программе обработки исключительной ситуации

В начале этой главы говорилось о причине использования исключительных ситуаций — упрощение клиентской программы через разделение основного процесса обработки от обработки исключительных ситуаций. В данном примере причина эта была в лучшем случае вторичной. Клиентская программа замусорена оператором try и конструкторами catch с их параметрами и скобками.

Покажем еще один способ проектирования с исключительными ситуациями: исключительные ситуации генерируются там, где можно обнаружить ошибку и собрать данные, необходимые для ее исправления. Оператор catch помещается там, где можно принять решение о том, как исправить ошибку. В этом простом примере такое решение заключалось в простом пропуске отображения ответа.

Обозначение исключительной ситуации

Обозначение исключительных ситуаций состоит в определении того, какие исключительные ситуации могут быть сгенерированы в рамках этой функции. Если функция не отслеживает саму исключительную ситуацию и ожидает, что другая функция разрешит эту проблему, необходимо объявить исключительную ситуацию.

Зарезервированное слово throw используется при обозначении исключительных ситуаций. Его общая синтаксическая форма объединяет обычное объявление функции, зарезервированное слово throw и список типов, значения которых генерируются функцией при поиске программы обработки исключительной ситуации.

```
functionDeclaration throw (Type1, Type2, ... TypeN);
```

Исключительные ситуации могут быть сгенерированы программой функции неявно, когда недопустимое условие возникает при вызове функцией своей серверной функции, или явно, используя зарезервированное слово throw.

Если исключительная ситуация генерируется программой функции и отслеживается самой функцией, не требуется включать ее в список throw. Если серверная функция генерирует исключительную ситуацию и отслеживает ее, то эта исключительная ситуация не должна включаться в список. В списке располагаются только те исключительные ситуации, которые должен обрабатывать клиент.

Например, функция inverse() в листинге 18.4 генерирует (и не отслеживает) две исключительные ситуации явно — символьный массив и long. Определение этой функции должно включать зарезервированное слово throw с такими двумя типами.

```
inline void inverse(long value, double& answer)
           throw (char*, long)
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
     throw MSG::msg(1);                                  // явный throw
  if (value < 0)
     throw value; }                                      // явный throw
```

Подобным же образом, функция fraction() в листинге 18.4 не генерирует каких-либо явных исключительных ситуаций, но ее серверная функция inverse() генерирует (и не отслеживает) две исключительные ситуации. То есть функция fraction() генерирует две исключительные ситуации неявно и должна обозначить их обе.

```
inline void fraction (long numer, long denom, double& result)
           throw (char*, long)
{ inverse(denom, result);                                // неявный throw
  result = numer * result; }                             // result = numer/denom
```

Если функция не генерирует исключительные ситуации, она может быть объявлена с пустой спецификацией throw(). Например:

```
void foo() throw ();          // ожидается отсутствие исключительных ситуаций
```

Если функция не определяет спецификацию исключительной ситуации, она может сгенерировать любую исключительную ситуацию.

```
void foo();      // throw отсутствует: ожидаются любые исключительные ситуации
```

Хорошо, если бы обозначение исключительной ситуации, которую функция фактически не сгенерировала, было бы в C++ ошибкой. Также было бы хорошо, если бы отсутствие обозначения исключительной ситуации, сгенерированной функцией, явно или неявно было бы ошибкой. Однако это не так, и можно использовать обозначения, вводящие в заблуждение (обозначая исключительные ситуации, которые функция не генерировала), или несоответствующие обозначения (обозначая только части исключительных ситуаций, которые генерирует функция). См. листинг 18.4.

Обозначение исключительных ситуаций — мощный метод документирования структуры программы. Убедитесь, что он используется разумно.

```
Enter numerator and positive
denominator (any letter to quit): 11 0

Zero denominator is not allowed

Enter numerator and positive
denominator (any letter to quit): 11 -11

Negative denominator: -11

Value of the fraction:       -1

Enter numerator and positive
denominator (any letter to quit): -11 44
Value of the fraction:       -0.25

Enter numerator and positive
denominator (any letter to quit): quit
```

Рис. 18.3. *Вывод для программы из листинга 18.5*

Когда функция обрабатывает исключительные ситуации только частично, это отражается в том, как функция обозначает исключительные ситуации. В листинге 18.5 показано обозначение исключительных ситуаций для различного разделения обязанностей между функциями inverse() и fraction(). Поскольку функция inverse() генерирует (и обозначает) те же самые исключительные ситуации, что и в листинге 18.4, функция fraction() сама обрабатывает исключительную ситуацию типа long. Следовательно, она обозначает только одну исключительную ситуацию в своем интерфейсе,— символьный массив.

Функция main() должна обрабатывать только одну исключительную ситуацию, а не две, как в листинге 18.4. Вывод примера выполнения программы представлен на рис. 18.3.

Листинг 18.5. Пример обозначения, генерации и отслеживания исключительных ситуаций

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
      static char* data [ ];                          // внутренние статические данные
public:
      static char* msg(int n)                         // общедоступный статический метод
      { if (n<1 || n > 5)                             // проверка допустимости индекса
           return data[0];
        else
           return data[n]; }                          // возвращение допустимой строки
} ;

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",              // хранилище текста
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction: "
} ;

inline void inverse(long value, double& answer)
           throw (char*, long)
{  answer = (value) ? 1.0/value : DBL_MAX;
   if (answer==DBL_MAX)
      throw MSG::msg(1);
   if (value < 0)
      throw value; }

inline void fraction (long numer, long denom, double& result)
           throw (char*)
{ try {
     inverse(denom, result); }                        // result = 1.0 / denom
   catch (long val)                                   // отрицательное значение - OK
   { cout << MSG::msg(2) << val << "\n\n"; }
   result = numer * result; }                         // result = numer / result

int main()
{  while (true)
   { long numer, denom; double ans;                   // числитель / знаменатель
     cout << MSG::msg(3) << MSG::msg(4);              // запрос ввода данных от пользователя
     if ((cin >> numer >> denom) == 0) break;         // ввод данных
     try {
        fraction(numer,denom,ans);                    // вычисление ответа
        cout << MSG::msg(5) << ans <<"\n\n"; }        // допустимый ответ
    catch (char* str)                                 // нулевой знаменатель
    { cout << str; }
   }
  return 0;  }
```

В этом примере показано преимущество обозначения исключительных ситуаций в интерфейсах функций. Когда программист клиентской части желает знать, какие исключительные ситуации должны обрабатываться клиентской функцией, достаточно проверить обозначения всех серверных функций, которые вызываются этой клиентской функцией.

Повторная генерация исключительной ситуации

Обратите внимание на то, что поведение программы, показанное на рис. 18.3, отличается от поведения, представленного на рис. 18.2. На рис. 18.2 отрицательное значение знаменателя отклоняется, и у пользователя запрашивается новый ввод. На рис. 18.3 отрицательное значение знаменателя отклоняется, но значение результата печатается в любом случае.

Причина этого в том, что функция `fraction()` сама осуществляет обработку исключительной ситуации (посредством вывода сообщения и значения знаменателя), а функция `main()` предполагает, что результат является допустимым, и не подавляет его вывод.

Это обычная ситуация, когда функция может обработать исключительную ситуацию только частично, но требуется предпринять некоторые другие действия в одной из вызывающих ее программ. C++ поддерживает такую потребность и разрешает функции выполнить повторную генерацию исключительной ситуации. Используйте оператор `throw` в структуре `catch`.

Например, функция `inverse()` может избежать ввода в заблуждение функции `main()`, которая предполагает, что она завершила восстановление, генерируя снова исключительную ситуацию.

```
inline void fraction (long numer, long denom, double& result)
           throw (char*, long)                  // обозначение дополнительной
                                                // исключительной ситуации
{ try {
    inverse(denom, result); }                   // result =1.0 / denom
  catch (long val)
       { cout << MSG::msg(2) << val << "\n\n";
         throw val; }                           // повторная генерация
  result = numer * result; }
```

Обратите внимание, что не возникает бесконечный цикл. Исключительная ситуация, сгенерированная в области видимости структуры `catch`, не может попасть в эту область видимости. Для этого исключительная ситуация должна происходить из блока `try`, который предшествует структуре `catch`. Формально исключительная ситуация рассматривается как обработанная при передаче ее программе обработки исключительной ситуации. Следовательно, этот оператор `throw` будет осуществлять поиск другой программы обработки ошибки `long` на более высоком уровне в клиентской программе, которая вызвала функцию `fraction()`.

Другой способ повторной генерации исключительной ситуации того же самого типа (и значения) — указать `throw` в структуре `catch`, и исключительная ситуация, определенная в параметре структуры `catch`, будет повторно сгенерирована.

```
inline void fraction (long numer, long denom, double& result)
           throw (char*, long)                  // обозначение дополнительной
                                                // исключительной ситуации
{ try {
    inverse(denom, result); }                   // result = 1.0 / denom
  catch (long val)
       { cout << MSG::msg(2) << val << "\n\n";
         throw; }                               // то же, что и "throw val"
  result = numer * result; }
```

В листинге 18.6 представлен этот метод. Функция `inverse()` та же, что и в листинге 18.5. Функция `fraction()` выполняет частичную обработку исключительной ситуации `long`, но затем генерирует ее снова. `Fraction()` должна затребовать эту исключительную ситуацию в своем интерфейсе, а `main()` должна обеспечить

```
Enter numerator and positive
denominator (any letter to quit): 11 0

Zero denominator is not allowed

Enter numerator and positive
denominator (any letter to quit): 11 -11

Negative denominator: -11

Enter numerator and positive
denominator (any letter to quit): -11 44
Value of the fraction:       -0.25

Enter numerator and positive
denominator (any letter to quit): quit
```

Рис. 18.4. *Вывод для программы из листинга 18.6*

оператор `catch` для обработки исключительной ситуации. Если `main()` не в состоянии сделать это, программа завершается аварийно.

Единственная цель повторной генерации данной исключительной ситуации — избежать отображения результата в `main()`. Следовательно, отсутствует обработка, которую структура `catch` должна выполнить в `main()`. Именно поэтому тело блока `catch` — пустое. Оно все еще должно находиться здесь. Чтобы избежать генерации предупреждения, что параметр структуры `catch` не используется, он был пропущен в списке параметров и оставлен только тип значения. Это допустимый метод C++, хотя и немного громоздкий.

Вывод для программы показан на рис. 18.4. Видно, что посторонний вывод подавляется.

Листинг 18.6. Пример повторной генерации исключительной ситуации в структуре `catch`

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
      static char* data [];                    // внутренние статические данные
public:
      static char* msg(int n)                  // общедоступный статический метод
      { if (n<1 || n > 5)                      // проверка допустимости индекса
           return data[0];
        else
           return data[n]; }                   // возвращение допустимой строки
} ;

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",       // хранилище текста
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction: "
} ;

inline void inverse(long value, double& answer)
           throw (char*, long)
{  answer = (value) ? 1.0/value : DBL_MAX;
   if (answer==DBL_MAX)
      throw MSG::msg(1);
   if (value < 0)
      throw value; }

inline void fraction (long numer, long denom, double& result)
           throw (char*, long)
{  try {
     inverse(denom, result); }              // result - 1.0 / denom
   catch (long val)                         // отрицательное значение - OK
     { cout << MSG::msg(2) << val << "\n\n";
       throw val; }
   result = numer * result; }               // result = numer / denom
```

```
int main()
{  cout << endl << endl;
   while (true)
   { long numer, denom; double ans;                    // числитель / знаменатель
     cout << MSG::msg(3) << MSG: :msg(4) ;             // запрос ввода данных пользователем
     if ((cin >> numer >> denom) == 0) break;          // ввод данных
     try {
          fraction (numer,denom,ans);                  // вычисление ответа
          cout << MSG::msg(5) << ans <<"\n\n"; }       // действительный ответ
     catch (char* str)                                 // нулевой знаменатель
     { cout << str; }
     catch (long)                                      // просто тип
        { }                                            // empty body
    }
   return 0;  }
```

Это мощный метод объединения нескольких функций для обработки одной исключительной ситуации. Используйте его аккуратно, поскольку в основе этого подхода лежит разделение (обработка исключительной ситуации) того, что, возможно, составляет одно целое. Когда трудно сосредоточить обработку исключительной ситуации в одном месте, программисты могут попытаться использовать этот метод, чтобы упростить написание программ. Вероятно, программа станет более сложной для понимания.

Исключительные ситуации с объектами класса

В приведенных выше примерах операторы `throw` передают блоку `catch` управление, а также значение конкретного типа. Доступ к этому значению можно осуществлять в блоке `catch`. Такой метод помогает установить связи между местом обнаружения и местом исправления ошибки.

Пересылка значения конкретного типа является как привилегией (устанавливается связь), так и ограничением, поскольку функция не может генерировать значения того же типа, чтобы они обрабатывались различными блоками `catch`. Например, если функция генерирует две различные символьные строки из двух различных мест, они должны обрабатываться одним и тем же блоком `catch`. Если исправление ошибки ограничивается выводом сообщения, блок `catch` выведет два разных сообщения.

```
void foo() throw (char*)
{ if (test1())
    throw "One bad thing happened";          // одна проблема
  else if (test2())
    throw "Another bad thing happened";      // другая проблема
  proceed_safely(); }                        // все в порядке

void client()
{ try
    { foo(); }                               // все в порядке
   catch(char* msg)
   { cout << msg << endl; } }                // любая из двух проблем
```

Если поведение программы должно отличаться для разных источников проблем, то этот механизм передачи данных становится слишком ограниченным — блок `catch` должен проанализировать данные, отправленные оператором `throw`, и выбрать разные ветви в зависимости от результата. Здесь надо знать цель обработки разных ошибок в различных блоках `catch`.

Другое собственное ограничение этого механизма обработки исключительных ситуаций состоит в том, что из оператора try в блок catch может быть отправлено только одно значение данных. Когда необходимо переслать более одного значения данных, программист должен прибегнуть к уловке. В примерах, показанных в листингах 18.1–18.6, для обработки исключительных ситуаций для отрицательного значения знаменателя требуется две части информации: указание, что знаменатель отрицательный, и его значение. Одна часть информации (значение знаменателя) передается как параметр для блока catch, а для сообщения об ошибке используется глобальный символьный массив.

C++ разрешает эти проблемы, допуская генерацию составных объектов вместо простых значений встроенных типов.

Синтаксис объектов генерации, обозначения и отслеживания

Генерация объекта исключительной ситуации добавляет новое измерение для программирования на C++. Проектировщик должен решить, какие элементы данных направляются из места, где ошибка была обнаружена, к месту, где происходит ее исправление. Для каждой исключительной ситуации создайте класс, объекты которого могут нести необходимые данные от места доступа к данным объекта. Методы этого класса позволят структуре catch иметь соответствующий доступ к данным объекта.

Например, класс ZeroDenom можно спроектировать для передачи данных о нулевом знаменателе. По месту обнаружения ошибки объект такого класса будет создан и передан. Для него требуется только одна часть информации, которая будет одинаковой для всех случаев появления ошибок. Следовательно, класс ZeroDenom должен содержать конструктор по умолчанию. В блоке catch выводится сообщение. Класс ZeroDenom может предоставить метод print(), который будет вызываться блоком catch.

```
class ZeroDenom {
  char *msg;          // данные должны передаваться программе обработки ошибки
public:
  ZeroDenom ()                  // вызывается оператором throw
  { msg = MSG::msg (1); }
  void print () const           // вызывается в блоке catch
  { cout << msg; }
} ;
```

Для использования объектов класса в качестве носителей информации об исключительной ситуации необходимо пройти те же три этапа:

1) генерация исключительной ситуации

2) отслеживание исключительной ситуации

3) обозначение исключительной ситуации

Функция, обнаруживающая состояние исключительной ситуации, например inverse(), создает объект этого класса и отправляет его на поиск блока catch.

```
if (answer==DBL_MAX)
    throw ZeroDenom();                  // необычный синтаксис
```

Обратите внимание на синтаксис вызова конструктора по умолчанию с указанием имени класса и двух пустых скобок. В других контекстах (например, создание объекта в операторе new) использование скобок было бы синтаксической ошибкой. В этом контексте синтаксической ошибкой является отсутствие скобок. Если в отношении подобного синтаксиса возникают сомнения, то можно создать объект

требуемого типа, а затем сгенерировать его так, как генерируются переменные встроенных типов.

```
if (answer==DBL_MAX)
   { ZeroDenom zd; throw zd; }            // обычный синтаксис
```

Когда вместо конструктора по умолчанию используется другой конструктор, синтаксис генерации объекта такой же, как и для других контекстов. Например, для передачи информации об отрицательном знаменателе можно спроектировать класс NegativeDenom с элементами данных для сообщения об ошибке и значения знаменателя и с методами, осуществляющими доступ к элементам данных объекта.

```
class NegativeDenom {
       long val;       // закрытые данные информации исключительной ситуации
       char* msg;
public:
  NegativeDenom(long value)          // конструктор преобразования
       : val (value), msg (MSG::msg(2)) { }
  char* getMsg( ) const
    { return msg; }
  long getVal() const                // общедоступный метод доступа к данным
    { return val; }  } ;
```

Чтобы сгенерировать объект этого типа, конструктору требуется сообщить значение параметра при помощи метода, который генерирует объект, например inverse().

```
if (value < 0)                          // анализ ситуации
    throw NegativeDenom(value);         // генерация исключительной ситуации
```

Подобно объектам, не имеющим параметров, этот объект можно создать, воспользовавшись обычным синтаксисом, а затем сгенерировать его.

```
if (value < 0)
   { NegativeDenom nd(value); throw nd; }
```

Синтаксис обозначения исключительных ситуаций такой же, как для встроенных значений, но имя класса должно использоваться вместо имени встроенного типа. Вот функция inverse(), которая требует исключительных ситуаций класса ZeroDenom и класса NegativeDenom.

```
inline void inverse(long value, double& answer)
    throw (ZeroDenom, NegativeDenom)   // обозначение исключительной ситуации
{ answer = (value) ? 1.0/value : DBL_MAX;
  if (answer==DBL_MAX)
     throw ZeroDenom();                // генерация объекта класса
  if (value < 0)
     throw NegativeDenom(value); }     // генерация объекта класса
```

Чтобы отследить объект класса, его параметр должна определить структура catch. В рамках области видимости структуры catch правила доступа к объекту те же, что и для объектов любого другого класса. Покажем, как клиент main() отслеживает две исключительные ситуации.

```
try {
  fraction(numer,denom,ans);                    // вычисление ответа
  cout << MSG::msg(5) << ans <<"\n\n"; }        // действительный ответ
catch (const ZeroDenom& zd)                     // нулевой знаменатель
  { zd.print(); }
catch (const NegativeDenom &nd)                 // отрицательное значение
  { cout << nd.getMsg() << nd.getVal() << "\n\n"; }
```

Первая структура catch отправляет сообщение объекту, запрашивая от него вывод информации, а вторая структура catch извлекает значения элемента данных объекта, а затем выводит их на печать. Первый метод лучше. Во втором случае элементы данных класса NegativeDenom также могут быть общедоступными.

В листинге 18.7 показана та же программа, что и в листингах 18.1—18.6. Функция inverse() генерирует объекты класса ZeroDenom и NegativeDenom. Поскольку эти функции вызывает функция fraction(), не знающая способа обработки этих исключительных ситуаций, имейте в виду, что это та функция, которая генерирует исключительные ситуации. Функция fraction() также обозначает данные исключительные ситуации. Кроме того, main() должна поместить вызов fraction() в блок try и предоставить две структуры catch, по одной для каждой исключительной ситуации.

Листинг 18.7. Пример генерации объектов класса вместо встроенных значений

```
#include <iostream>
#include <cfloat>
using namespace std;

class MSG {
      static char* data [];                    // внутренние статические данные
public:
      static char* msg(int n)                  // общедоступный статический метод
      { if (n<1 || n > 5)                      // проверка допустимости индекса
           return data[0];
        else
           return data[n]; }                   // возвращение допустимой строки
} ;

char* MSG::data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",       // хранилище текста
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction: "
} ;

class ZeroDenom {
      char *msg;                          // данные, передаваемые программе обработки ошибок
public:
      ZeroDenom ()                            // вызывается оператором throw
      { msg = MSG::msg(1); }
      void print () const                     // вызывается в блоке catch
      { cout << msg; }
} ;

class NegativeDenom {
  long val;                                   // закрытые данные для исключительной ситуации
  char* msg;
public:
  NegativeDenom(long value)                   // конструктор преобразования
        : val(value), msg(MSG::msg(2)) { }
  char* getMsg() const
    { return msg; }
  long getVal() const                         // общедоступные методы доступа к данным
    { return val; }
} ;
```

```
inline void inverse(long value, double& answer)
          throw (ZeroDenom, NegativeDenom)
{  answer = (value) ? 1.0/value : DBL_MAX;
   if (answer==DBL_MAX)
      throw ZeroDenom();
   if (value < 0)
      throw NegativeDenom(value); }

inline void fraction(long numer, long denom, doubles result)
          throw (ZeroDenom, NegativeDenom)
{  inverse (denom, result);                          // result = 1.0 / denom
   result = numer * result; }                        // result = numer / denom

int main()
{
   while (true)
   { long numer, denom; double ans;                  // числитель/знаменатель
     cout << MSG::msg(3) << MSG::msg(4);             // запрос ввода данных пользователем
     if ((cin >> numer >> denom) == 0) break;        // ввод данных
     try {
        fraction(numer,denom,ans);                   // вычисление ответа
        cout << MSG::msg(5) << ans <<"\n\n"; }       // действительный ответ
     catch (const ZeroDenom& zd)                     // нулевой знаменатель
     { zd.print() ; }
     catch (const NegativeDenom &nd)                 // отрицательное значение
     { cout << nd.getMsg() << nd.getVal() << "\n\n"; }
    }
   return 0;
   }
```

Использование наследования с исключительными ситуациями

Состояния ошибки в программе могут быть подобными друг другу. Информация, которая необходима для исправления ошибки, может иметь такую же структуру. Например, в листинге 18.7 каждая исключительная ситуации передает указатель на символьный массив, который должен быть выведен как сообщение об ошибке.

Как часто случается с подобными классами, проектировщик может организовать классы исключительных ситуаций программы в иерархии наследования. Например, можно доработать классы ZeroDenom и NegativeDenom так, что класс NegativeDenom порождается из класса ZeroDenom.

```
class ZeroDenom {
protected:
        char *msg;
public:
        ZeroDenom (char* message) : msg(message)
        { }
        void print () const
        { cout << msg; }
} ;
```

В предыдущей версии этого класса имя символьного массива было жестко запрограммировано в конструкторе класса. В результате клиент этого класса, функция inverse() в листинге 18.7, не должна была знать, какое сообщение отправить

исключительной ситуации. Функция должна создавать объект исключительной ситуации, используя конструктор по умолчанию. В этом варианте имеется класс ZeroDenom, который не знает, что передают его объекты. Его клиенты должны будут определять явно, какое сообщение передавать.

Трудно сказать, какой подход лучше. Как правило, первый подход (реализованный в листинге 18.7) передает обязанности вниз к серверному классу ZeroDenom, а второй подход — вверх клиентам класса. Однако общая схема распределения информации между классами программы может сделать более привлекательным второй подход. Хотелось бы быть уверенным, что это различие запомнилось, что оно понятно и вполне осознается в программе.

```
class NegativeDenom : public ZeroDenom {
       long val;
public:
       NegativeDenom(char *message, long value)
           : ZeroDenom(message), val(value) { }
       void print () const
       { cout << msg << val << "\n\n"; }
} ;
```

NegativeDenom был порожден из ZeroDenom. Можно ли породить ZeroDenom из NegativeDenom? В принципе возможно. С практической точки зрения, однако, это не очень хорошая идея. Класс NegativeDenom содержит больше компонентов в наборе данных, чем класс ZeroDenom.

Данные в базовом классе ZeroDenom объявляются защищенными, а не закрытыми, так что порожденный класс NegativeDenom способен осуществить доступ к базовым данным. Если же данные ZeroDenom были бы закрытыми, то методы в NegativeDenom должны использовать методы ZeroDenom для доступа к данным ZeroDenom. Например, можно спроектировать класс NegativeDenom.

```
class NegativeDenom : public ZeroDenom {
       long val;
public:
       NegativeDenom(char *message, long value)
           : ZeroDenom(message), val(value) { }
       void print () const
       { ZeroDenom::print();                    // вызов базового метода
         cout << val << "\n\n"; }
} ;
```

С одной стороны, предполагалось, что если два алгоритма, в базовом классе и в производном классе, содержат общие элементы, было бы замечательно подчеркнуть этот факт в программе производного класса. Для этого надо вызвать метод базового класса в соответствующем методе производного класса. С другой стороны, добавление к базовому классу методов доступа, которые используются только в производном классе, это напрасная трата времени.

Когда классы исключительных ситуаций связаны наследованием, обозначение исключительной ситуации и ее генерация объектов такие же, как и для несвязанных классов исключительной ситуации. Тем не менее отслеживание исключительных ситуаций может вызывать дополнительные проблемы, если только не будет уделено достаточное внимание взаимоотношениям между классами. В листинге 18.8 показана программа из листинга 18.7, измененная таким образом, что класс NegativeDenom является производным от класса ZeroDenom.

Функции inverse() и fraction() требуют такие же исключительные ситуации, как в листинге 18.7. Однако именно функция inverse(), а не классы исключительных ситуаций ZeroDenom и NegativeDenom, знает, какое сообщение генерируется для каждой исключительной ситуации.

```
Enter numerator and positive
denominator (any letter to quit): 11 0

Zero denominator is not allowed

Enter numerator and positive
denominator (any letter to quit): 11 -42

Negative denominator: Enter numerator and positive
denominator (any letter to quit): -11 44
Value of the fraction:       -0.25

Enter numerator and positive
denominator (any letter to quit): exit
```

Рис. 18.5. *Вывод для программы из листинга 18.8*

Пример вывода для этой программы приведен на рис. 18.5. В нем использована приблизительно та же последовательность входных данных, что и для предыдущего варианта программы.

Как можно видеть, вывод программы неверен. Когда знаменатель отрицательный, программа выводит соответствующее сообщение об ошибке, но не отображает значение отрицательного знаменателя. Вместо этого она переходит к запросу следующего набора входных данных. Что сделано неправильно?

Вспомним, что исключительную ситуацию можно сгенерировать с двумя контекстами: из блока `try` и вне любого блока `try`. Когда исключительная ситуация генерируется вне блока `try`, функция завершает свое выполнение немедленно и проверка повторяется в вызывающей программе. Вызов функции, которая генерирует исключительную ситуацию, может быть либо в рамках блока `try`, либо вне блоков `try`.

Например, функция `inverse()` генерирует свои исключительные ситуации вне какого-либо блока `try`. Когда генерируется любая из этих исключительных ситуаций, выполнение `inverse()` немедленно прекращается и управление передается вызывающей ее программе `fraction()`. В программе `fraction()` вызов `inverse()`, которая генерирует исключительную ситуацию, находится вне любого блока `try`. Именно поэтому `fraction()` также немедленно завершается и управление передается `main()`.

Листинг 18.8. Пример использования классов исключительных ситуаций, связанных наследованием

```
#include <iostream>
# include <cfloat>
using namespace std;

class MSG {                                    // внутренние статические данные
      static char* data [];
public:
      static char* msg(int n)                  // общедоступный статический метод
      { if (n < 1 || n > 5)                    // проверка допустимости индекса
           return data[0];
        else
           return data[n]; }                   // возвращение допустимой строки
} ;

char* MSG:: data [] = { "\nBad argument to msg()\n",
"\nZero denominator is not allowed\n\n",       // область хранения текста
"\nNegative denominator: ",
"Enter numerator and positive\n",
"denominator (any letter to quit): ",
"Value of the fraction: "
} ;

class ZeroDenom {
protected:
     char *msg;
public:
     ZeroDenom (char* message) : msg(message)
     { }
```

```
      void print () const
      { cout << msg; }
} ;

class NegativeDenom : public ZeroDenom {
      long val;
public:
      NegativeDenom(char *message, long value)
         : ZeroDenom(message), val(value) { }
      void print () const
      { cout << msg << val << "\n\n"; }
} ;

inline void inverse(long value, double& answer)
         throw (ZeroDenom, NegativeDenom)
{  answer = (value) ? 1.0/value : DBL_MAX;
   if (answer==DBL_MAX)
      throw ZeroDenom(MSG::msg(1));
   if (value < 0)
      throw NegativeDenom(MSG::msg(2) , value); }

inline void fraction (long numer, long denom, double& result)
         throw (ZeroDenom, NegativeDenom)
{  inverse (denom, result);                          // result = 1.0 / denom
   result = numer * result; }                        // result = numer / denom

int main()
{
   while (true)
   { long numer, denom; double ans;                  // числитель/знаменатель
     cout << MSG::msg(3) << MSG::msg(4);             // запрос пользователю на ввод данных
     if ((cin >> numer >> denom) == 0) break;        // ввод данных
     try {
          fraction(numer,denom,ans);                 // вычисление ответа
          cout << MSG: :msg(5) << ans <<"\n\n"; }    // допустимый ответ
     catch (const ZeroDenom &zd)                     // нулевой знаменатель
     { zd.print(); }
     catch (const NegativeDenom &nd)                 // отрицательное значение
     { nd.print(); } }                               // конец цикла
   return 0;
   }
```

Когда исключение генерируется внутри блока try, управление передается в конец блока, который содержит оператор throw. После блока try должны размещаться один или несколько структур catch. Параметры этих структур catch проверяются друг за другом. Если совпадение не обнаруживается, то ситуация рассматривается так же, как если бы исключительная ситуация была сгенерирована вне блока try. Выполнение функции немедленно прекращается, а управление передается вызвавшей ее программе. Если находится совпадение, поиск прекращается и управление передается совпадающей структуре catch. После завершения выполнения этой структуры catch все последующие структуры catch пропускаются. Выполнение продолжается с помощью обработки оператора, который располагается после структур catch.

Типы совпадают, если они одинаковы. Они также совпадают, если производный объект порождается из "пойманного" типа или если сгенерированный объект ссылается на объект производного класса, тогда как "пойманный" тип указывает

на объект базового класса. Вспомните правило: объект производного класса может использоваться там, где ожидается объект базового класса (оно подробно рассматривается в главе 15).

В листинге 18.8 при обработке исключительной ситуации NegativeDenom выполнение функций inverse() и fraction() прекращается, поскольку они не содержат блок try. Когда заканчивается выполнение функции fraction(), она генерирует исключительную ситуацию (полученную из inverse()) для функции main(). Поскольку main() вызывает fraction() в блоке try, структуры catch проверяются друг за другом. Вначале проверяется "ловушка" с параметром ZeroDenom. Поскольку объект NegativeDenom, сгенерированный fraction(), может использоваться там, где ожидается объект ZeroDenom, поиск прекращается и выполняется блок-"ловушка" ZeroDenom. Он отправляет сообщение базового класса ZeroDenom::print() его объекту-аргументу. Он выводит только сообщение, а не значение, которое содержит объект NegativeDenom, но ZeroDenom::print() не знает, как выполнить вывод, поскольку значение является элементом данных производного класса.

Некоторые компиляторы могут выдавать предупреждение о проблеме. Однако нет такого компилятора, который пометит эту структуру как синтаксическую ошибку, поскольку неотъемлемым правом программистов является размещение блоков-"ловушек" в том порядке, который кажется им наиболее подходящим.

Следовательно, не следует помещать первым блок-"ловушку" для базового класса. Необходимо разместить его последним. Вот как будет выглядеть функция main() из листинга 18.8 после исключения этой проблемы.

```
int main()
{ while (true)
  { long numer, denom; double ans;
    cout << MSG::msg(3) << MSG::msg(4);          // приглашение ввода данных
                                                 // пользователем
    if ((cin >> numer >> denom) == 0) break      // ввод данных
    try {
       fraction(numer,denom,ans);                // вычисление ответа
       cout << MSG::msg(5) << ans <<"\n\n"; }    // допустимый ответ
    catch (const NegativeDenom &nd)              // производный класс
    { nd.print() ; }
    catch (const ZeroDenom &zd)                  // базовый класс
    { zd.print(); } }                            // конец цикла
  return 0; }
```

Стандартная библиотека исключительных ситуаций

Стандартная библиотека C++ определяет несколько классов стандартных исключительных ситуаций, организованных с соблюдением иерархии наследования. Наиболее важными классами являются класс exception (все буквы в нижнем регистре), который представляет собой базовый класс в иерархии, и bad_alloc, который порождается из класса exception.

Класс exception определяется в заголовочном файле <exception>, <except.h> или <exception.h>. Класс exception включает виртуальную функцию what(), которая возвращает символьный указатель, подобно методу getMsg() в классе из листинга 18.7, приведенного ранее. Информационное наполнение строки не определено, но можно спроектировать класс, являющийся наследником класса. В этом классе можно переопределить what().

```
class NegativeDenom {
    long val;                  // закрытые данные для информации exception
    char* msg;
```

```
public:
    NegativeDenom(long value)        // конструктор преобразования
          : val(value), msg(MSG::msg(2)) { }
    const char* what() const         // может возвратить произвольную строку
       { return msg; }
    long getVal () const
       { return val; }
} ;
```

Класс bad_alloc определяется в заголовочном файле <new> или <new.h>. Его объект генерируется, когда оператор new не может выделить требуемый объем памяти из динамически распределяемой области. Пока не все компиляторы поддерживают эту исключительную ситуацию. Приведем небольшой пример, в котором строится длинный связанный список блоков памяти. Он использует исключительную ситуацию bad_alloc. Кроме того, он проверяет, возвращает ли оператор new пустой указатель.

```
#include <iostream>                                  // включая файлы
#include <exception>
#include <new>
using namespace std;

struct Block
{ char a[1000];                                      // блок памяти
  Block* next;
Block (Block* ptr)                                   // присоединить перед ptr
  { next = ptr; } }

int main()
{ Block *list = 0, *p; int cnt = 1;
  while (true)                          // перейти, пока он не завершится аварийно
  { try {
       p = new Block(list) ; }                       // это не выполнится
    catch (bad_alloc &bad)
    {  cout << bad.what() << endl;                   // сообщение при исправлении
       exit(0); }
    if (p == 0)                                      // сообщение при исправлении
       { cout << "Out of memory\n\n"; exit(0); }
    list = p;                                        // успех : верх списка
    if (++cnt%100 == 0)
       cout << "Block #" << cnt << endl; }          // выполнение отслеживания
  while (p != 0)
  { p = p->next; delete list; list = p; }            // освобождение памяти
  return 0; }
```

Механизм исключительных ситуаций не поддерживает асинхронные исключительные ситуации, например прерывания. Он обрабатывает синхронные исключительные ситуации, возникающие в процессе последовательного выполнения, например переполнение, ошибки выхода за пределы области, ошибки выделения ресурсов и неверные входные данные. Исключительные ситуации не должны использоваться для состояний, которые являются обычными для потока выполняемых операций, например завершение выполнения одного этапа обычной обработки (окончание списка цикла) и начало другого.

Использование исключительных ситуаций языка C++ имеет два главных преимущества. Во-первых, они обеспечивают обмен информацией между местом обнаружения ошибки и местом, где можно ее исправить. Во-вторых, возврат стека в исходное состояние в процессе завершения вызванной функции и передача

управления обратно вызывающей функции являются безопасными. Если любая из вызванных функций размещает объекты в стеке, то их деструкторы вызываются таким образом, как если бы возврат к каждой из этих функций выполнялся обычным способом. Возвращаются системные ресурсы и не появляются взаимоблокировки и расходы ресурсов.

Операции приведения типов

Данный материал фактически не относится к этой главе. Однако его невозможно было обсудить ранее, поскольку он основывается на расширенных понятиях наследования, шаблонов и обработки исключительных ситуаций.

Кроме того, решался вопрос о целесообразности обсуждения операций приведения. Они были добавлены в язык C++ относительно недавно и опыт их использования в программировании ограничен. Отсутствуют серьезные доказательства, что эти операции лучше, чем стандартные простые приведения, которые применялись раньше.

Однако операции приведения типов представляют собой набор интересных идей из области программной инженерии. Рекомендуем вам познакомиться с ними. Стоит ли их использовать на практике — решайте сами.

Операции приведения типов и конструкторы преобразования ослабляют систему строгого контроля за типами в языке C++. Они расширяют возможные преобразования типов. Программисты клиентской части и программисты, осуществляющие сопровождение, могут не знать, какие преобразования возможны и какие из них фактически выполняются.

Чтобы помочь программистам справиться с этой ситуацией, C++ вводит несколько дополнительных операций приведения. Область их действия шире, чем у стандартных операций приведения типов. Фактически это является одним из их преимуществ, поскольку операции приведения легче заметить в исходной программе, чем стандартные операции приведения типов.

Операция static_cast

Операция static_cast может применяться везде, где работает стандартное приведение типа. Она не будет использоваться там, где стандартное приведение рассматривается как слишком опасное. Представим несколько примеров.

static_cast является унарной операцией, т. е. применяемой к операнду одного типа для получения значения другого типа. Программист должен определить операнд (объект или выражение преобразуемого типа) в обычных скобках. Дополнительно программист должен определить тип назначения как параметр в угловых скобках, подобно синтаксису, используемому в шаблонах.

```
valueOfTargetType = static_cast<TargetType>(valueOfSourceType);
```

Как можно заметить, это приведение типа в действительности не является унарной операцией, поскольку для него требуется как значение исходного типа (один операнд), так и имя типа назначения (второй операнд). Однако это и не бинарная операция, потому что имя приведения типов не появляется между операндами, как происходит в бинарных операциях.

Использование такого приведения типов не ограничивается только присваиванием. Оно может применяться в любом месте, где может использоваться значение типа назначения TargetType. Вот простой пример.

```
double d; int i = 20;
d = static_cast<double>(i) ;                    // ok: d равно 20.0
```

Лучше ли это, чем старый и надежный друг для приведения типов — double? Это совершенно одинаковые вещи.

```
double d; int i = 20;
d = double(i) ;                    // ok: d равно 20.0
```

Рассмотрим сложный пример. Класс Account предусматривает несколько операций преобразования, которые извлекают значения своих компонентов. Для простоты используется массив фиксированного размера для имени владельца.

```
class Account {                                    // базовый класс иерархии
protected:
  double balance;                                  // защищенные данные
  int pin;                                         // идентификационный номер
  char owner[40];
public:
  Account(const char* name, int id, double bal)
  { strcpy(owner, name) ;                          // общий
    balance = bal; pin = id;}
  operator double () const                         // инициализация полей данных
  { return balance; }
  operator int () const
  { return pin; }
  operator const char* () const
  { return owner; }
  void operator -= (double amount)
  { balance -= amount; }
  void operator += (double amount)
  { balance += amount; }                           // безусловное приращение
} ;
```

Как уже говорилось в главе 15, эти функции перегруженных операций могут вызываться с использованием того же синтаксиса, что и в стандартных приведениях типов.

```
Account a1("Jones", 1122, 5000);                   // создать объект
int pin = (int)a1;
double bal = (double) a1;                          // допустимые приведения
const char *c = (const char*) a1;
```

Операция static_cast также действительна в этом контексте. Она выполняет то же самое, что и стандартные операции приведения типов.

```
Account a1("Jones", 1122, 5000);                   // создать объект
int pin = static_cast<int>(a1);                    // ok
double bal = static_cast<double> (a1);
const char *c = static_cast<const char*>(a1);
```

Не ошибитесь: операции static_cast работают только потому, что класс Account поддерживает перегруженные операции преобразования int, double и const char*. В противном случае попытка применения операции static_cast к объектам Account была бы так же напрасна, как и попытка стандартных приведений.

Основная разница между стандартными приведениями и static_cast состоит в том, как они осуществляют преобразование указателей. Стандартные приведения основываются на здравом смысле программистов. Если требуется, чтобы указатель двойной длины ссылался на переменную int, значит, имеется уважительная причина поступать подобным образом.

В листинге 18.9 представлен пример использования преобразований указателя. Результаты выполнения программы показаны на рис. 18.6.

При запуске main() два указателя pd и pi установлены для обозначения целой переменной i. Затем эти указатели разыменовываются, чтобы указывать на значение i. Целый указатель pi правильно отыскивает значение i, а указатель pd двойной длины извлекает мусор.

Затем указатель двойной длины pd устанавливается для указания на объект a1 класса Account. Разыменовывая этот указатель, программа не только извлекает значение элемента данных объекта balance, но и заменяет его на любое необходимое.

```
i=9.88131 e-323 i=20
balance = 5000
balance = 10000
```

Рис. 18.6. *Вывод для программы из листинга 18.9*

Листинг 18.9. Примеры преобразования указателя с использованием стандартных приведений

```
#include <iostream>
using namespace std;

class Account {                                    // базовый класс иерархии
protected:
   double balance;                                 // защищенные данные
   int pin;                                        // identification number
   char owner[40];
public:
   Account(const char* name, int id, double bal)   // общий
   { strcpy(owner, name);                          // инициализация полей данных
     balance = bal; pin = id; }
   operator double () const                        // общий для обоих счетов
   { return balance; }
   operator int () const
   { return pin; }
   operator const char* () const
   { return owner; }
   void operator -= (double amount)
   { balance -= amount; }
   void operator += (double amount)
   { balance += amount; }                          // безусловное приращение
} ;

int main()
{
   double *pd, d=20.0; int i = 20, *pi = &i;
   pd = (double*) pi;
   cout << "i=" << *pd << " i=" << *pi <<endl;
   Account a1 ("Jones", 1122,5000);                // создать объекты
   pd = (double*)(&a1);
   cout << "balance = " << *pd << endl;
   *pd =10000;                                     // изменить элемент данных
   cout << "balance = " << *pd << endl;
   return 0;
}
```

Здесь поведение static_cast отличается от поведения стандартных приведений. Указатель двойной длины способен неверно представить значение переменной i, потому что целый адрес может использоваться как операнд для приведения (double*). С помощью операции static_cast это сделать невозможно.

```
pd = (double*) pi;                    // ok
pd = static_cast<double*> (pi);       // синтаксическая ошибка
```

Указатель двойной длины pd в состоянии осуществить доступ и изменить элемент данных объекта Account только потому, что адрес Account может использоваться как операнд стандартного приведения. Если применяется операция static_cast, сделать это невозможно.

```
Account a1 ( "Jones", 1122, 5000);          // создать объект
pd = (double*)(&a1);                        // ok
*pd = 10000;                                // ok
pd = static_cast<double*>(&a1);             // синтаксическая ошибка
```

Это не означает, что операция static_cast не может использоваться с указателями. Она не может применяться с указателями, преобразование которых не имеет смысла с точки зрения программной инженерии. Рассмотрим, например, класс SavingsAccount, открыто порожденный из класса Account.

```
class SavingsAccount : public Account {
       double rate;                         // фиксированная процентная ставка
public:
       SavingsAccount(const char* name, int id, double bal)
           : Account(name, id, bal), rate (6.0) { }
       void payInterest()                   // платить раз в месяц
       { balance += balance * rate / 12 / 100; }
} ;
```

Объекты SavingsAccount могут выполнить все то же, что и объекты Account. Кроме того, они содержат больше элементов данных и больше функций-членов. Следовательно, указатель Account может ссылаться на объект SavingsAccount без каких-либо затруднений. Это безопасно и не требует какого-либо приведения, стандартного или любого другого.

```
Account a1("Jones", 1122, 5000);              // создать объекты
SavingsAccount a2("Smith", 1133, 3000);
Account *pa = &a2;          // сохранить преобразование, приведение не требуется
```

Указатель SavingsAccount не должен обозначать объект Account, потому что он может отправить объекту сообщение, на которое базовый объект, возможно, будет не в состоянии ответить.

```
SavingsAccount *psa = pa;                   // синтаксическая ошибка
```

Конечно, если указатель Account фактически ссылается на объект SavingsAccount, то присваивание (преобразование) имеет смысл. Однако сообщите об этом с помощью приведения.

```
psa = (SavingsAccount *)pa;                 // явное приведение все сделало
```

Это именно та ситуация, в которой операция static_cast использует свое неприятие указателей. Она может использоваться для данного преобразования вместо стандартного приведения.

```
psa = static_cast<SavingsAccount*>(pa);          // это просто замечательно
```

Операция static_cast может использоваться в тех ситуациях, где преобразование небезопасно. Здесь, например, указатель SavingsAccount устанавливается так, что ссылается на объект Account, а операция static_cast не согласна со способом выполнения операции стандартного приведения.

```
psa = static_cast<SavingsAccount*>(&a1);         // это просто замечательно
```

Обобщим преимущества этого приведения перед стандартным приведением. Во-первых, оно состоит из нескольких слов, благодаря чему его легче заметить в программе. Во-вторых, оно более требовательно, чем стандартные приведения.

Бьерн Страуструп, автор C++, говорил, что чем реже используются приведения, тем лучше, и все мешающее применению приведения является полезным.

Операция reinterpret_cast

Операция reinterpret_cast спроектирована для выполнения всего того, что могут делать стандартные операции приведения, но без ограничений, накладываемых операцией static_cast.

Операция reinterpret_cast может применяться, когда программист уверен, что компилятор не знает о фактических типах, на которые указывает указатель. В приведенном выше примере целый указатель p ссылается на значение двойной длины. В последней строке указатель двойной длины q присваивается значению p. Компилятору неизвестно, что указатель p фактически указывает на значение двойной длины. Программист сообщает это компилятору с помощью операции reinterpret_cast.

```
double y = 42;
int *p = reinterpret_cast<int*>(&y);          // потенциальная проблема
double *q = reinterpret_cast<double*>(p);     // p указывает на значение
                                              // двойной длины
cout << "The answer is " << *q << endl;       // выводит на печать 42!
```

Этого же результата можно достичь, используя стандартные приведения int* и double*.

```
double y = 42;
int *p = (int*)&y;                            // целый p указывает на значение
                                              // двойной длины: проблема
double *q = (double*)(p);                     // ok, потому что p указывает
                                              // на значение двойной длины
cout << "The answer is " << *q << endl;       // выводит на печать 42!
```

Считается, что операция reinterpret_cast лучше, чем стандартные операции приведения, потому что она более заметна.

Обратите внимание, что здесь не может использоваться static_cast. Она может преобразовывать значения различных типов, но не указатели. Кроме того, операция static_cast является переносимой, потому что компилятор проверяет, используются ли соответствующие типы (как числовые типы), операция преобразования или конструктор преобразования.

Операция reinterpret_cast не гарантирует свою переносимость. Она выбирает последовательность бит в исходном выражении и интерпретирует их в соответствии с правилами типа назначения.

Это приведение должно использоваться как можно реже. Если требуется применять приведение, используйте операцию reinterpret_cast, а не стандартное приведение.

Операция const_cast

Операция const_cast может аннулировать свойство объявления константой для значения или объекта-константы. Синтаксис ее такой же, как и у других современных операций приведения C++, включая указание типа назначения в угловых скобках и исходного выражения в скобках.

```
nonConstValue = const_cast<TypeName>(constValue);
```

Ее синтаксис и семантика более строгие, чем у других операций приведения. Все, что она может сделать,— это удалить свойство const у исходного значения constValue, поэтому становится возможным присвоение значения константы

constValue значению nonConstValue, которое не является константой. Тип nonConstValue должен быть именно TypeName, тип constValue — const TypeName.

Рассмотрим следующий пример. Поскольку переменная d определяется как const, обычный указатель не может на нее указывать.

```
const double d = 42;
double *pd = &d;                // ошибка: чтобы предотвратить *pd = 21
```

Указатель на значение-константу может указывать на переменную d, но он не меняет свое значение.

```
const double d = 42;
const double *pd = &d;          // ok, но не очень практично
*pd =21;                        // синтаксическая ошибка: указатель на const
```

Операция const_cast прибегает к уловке. Она удаляет требование константы и открывает возможность для изменения значения, которое определяется как const.

```
const double d = 42;
double *pd = const_cast<double*>(&d);        // удалить const
*pd =21;                                     // теперь все ok
cout << "The answer is " << *pd << endl;     // выводит 21
```

Это фокус, который невозможно сделать даже для приведения стандартных типов С. Единственной ситуацией, где это необходимо, является сопровождение программы, где переменная была определена как const в соответствии с новыми условиями. Вместо изменения существующего определения можно добавить новый код, где переменная меняется с использованием операции const_cast.

Использование операции const_cast удаляет защиту. Указатель не должен быть указателем на константу. Он может быть разыменован при изменении объекта, на который он указывает. Рассмотрим класс Account.

```
class Account {                                  // базовый класс иерархии
protected:
  double balance;                                // защищенные данные
  int pin;                                       // идентификационный номер
  char owner[40];
public:
  Account(const char* name, int id, double bal)      // общее
  { strcpy(owner, name);                         // инициализация полей данных
    balance = bal; pin = id; }
  operator double () const                       // общий для обоих счетов
  { return balance; }
  operator int () const
  { return pin; }
  operator const char* () const
  { return owner; }
  void operator -= (double amount)
  { balance -= amount; }
  void operator += (double amount)
  { balance += amount; }                         // безусловное приращение
} ;
```

При попытке вызова функции-члена, не константы (например, operator+=()) в объекте const класса Account компилятор отбросит этот код.

```
const Account a1("Jones", 1122, 5000. 0);      // создать объект
a1 += 1000.0;                                  // синтаксическая ошибка
```

При попытке установить обычный указатель Account так, чтобы он ссылался на объект const, компилятор отбросит этот код из опасения, что объект может измениться во время разыменования указателя.

```
const Account a1("Jones",1122,5000.0);      // создать объект
Account *pa = &a1;                          // синтаксическая ошибка
```

Если указатель ссылается на объект-константу, он должен быть определен как указатель на объект-константу. Это разрешается, но указатель не используется для изменения состояния объекта.

```
const Account a1("Jones",1122,5000.0) ;     // создать объект
const Account *pa = &a1;                    // ok
*pa += 1000.0;                              // синтаксическая ошибка
```

Однако обычный указатель можно установить так, чтобы он указывал на объект const, воспользовавшись операцией const_cast.

```
const Account a1("Jones",1122,5000.0);      // создать объект
Account *pa = const_cast<Account*>(&al);    // ok
*pa += 1000.0;                              // это допускается
```

В результате состояние объекта-константы изменяется. Его баланс теперь составляет $6000, но над объектом не была выполнена явная операция.

Единственное, что операция const_cast может сделать, это удалить защиту const. Она не может выполнить какой-либо дополнительный тип преобразований. Если тип nonConstValue (результат приведения) отличается от типа ConstValue, а преобразование типа необходимо, то оно должно быть выполнено как отдельный дополнительный шаг.

Например, операция преобразования const char*() класса Account возвращает символьный указатель, который не может (и не должен) использоваться для изменения содержимого символьного массива в объекте Account.

```
const Account a1("Jones",1122,5000.0);      // создать объект
char *c2 = static_cast<const char*>(a1);    // синтаксическая ошибка
```

Значение этого указателя может быть присвоено только указателю на константу. Такой указатель не применяется для изменения состояния объекта Account.

```
const Account a1("Jones",1122,5000.0);          // создать объект
const char *c2 = static_cast<const char*>(a1);  // это ok
strcpy(c2,"Jones");                             // синтаксическая ошибка
```

Использование операции const_cast в объекте Account не помогает, поскольку значение цели и исходное значение имеют разные типы.

```
const Account a1("Jones",1122,5000.0);      // создать объект
char *c2 = const_cast<char*>(a1);           // синтаксическая ошибка
```

Поскольку операция const_cast может выполнить только одно задание, совершенно правильно было бы вначале преобразовать объект-константу Account в указатель на константу, используя static_cast или стандартное приведение, а затем преобразовать указатель на константу в обычный указатель с помощью const_cast.

```
const Account a1("Jones",1122,5000.0);          // создать объект
const char *c1 = static_cast<const char*>(a1);  // это ok
char *c2 = const_cast<char*>(c1);               // и это ok
strcpy(c2,"Jones");                 // не является синтаксической ошибкой
```

Имя владельца изменяется без явной обработки объекта константы Account.

Операция dynamic_cast

Операция dynamic_cast является элементом набора компонентов C++, которые поддерживают информацию о типах в процессе исполнения (run-time-type information — RTTI). Остальными элементами этого набора компонентов являются операция typeid и структура type_info.

Операция dynamic_cast применяется для преобразования указателей (или ссылок) базового класса в указатели (или ссылки) одного из производных классов. Операция static_cast (или стандартное приведение) также может использоваться, но программа должна знать тип объекта, чтобы быть уверенной в том, что она преобразует указатель в соответствующий класс.

Операция dynamic_cast использует тот же синтаксис, что и другие операции приведения. Аргумент-указатель (или ссылка) заключается в круглые скобки, а тип назначения, в который преобразуется аргумент, указывается в угловых скобках. Если аргумент-указатель действительно принадлежит запрошенному типу назначения, операция возвращает аргумент-указатель на объект без изменений. Кроме того, она отправляет указатель на объект, если аргумент-указатель ссылается на объект класса, порожденного из типа назначения. В противном случае она возвращает нуль, и программа может проверить значение.

Чтобы этот метод работал, иерархия классов должна содержать как виртуальные, так и не виртуальные функции. Он не работает для наследования без виртуальных функций.

Рассмотрим упрощенный класс Account, который содержит виртуальную функцию display(). Она отображает содержимое объекта Account.

```
class Account {                                    // базовый класс иерархии
protected:
  double balance;                                  // защищенные данные
  int pin;                                         // идентификационный номер
  char owner[40];
public:
  Account(const char* name, int id, double bal)          // общий
  { strcpy(owner, name);                           // инициализация полей данных
    balance = bal; pin = id;}
  virtual void display()                           // виртуальная функция для RTTI
  { cout.setf(ios::fixed,ios::floatfield); cout.precision(2);
    cout <<setw(6) << pin << setw(20) << balance
         << " " << owner <<endl; }
  void operator -= (double amount)
  { balance -= amount; }
  void operator += (double amount)
  { balance += amount; }                           // безусловное приращение
} ;
```

Производный класс SavingsAccount добавляет метод payInterest(), являющийся дополнительным, и переопределяет базовый метод display() таким образом, что отображается дополнительный элемент данных interest.

```
class SavingsAccount : public Account {
       double rate, interest;                          // накопленные проценты
public:
  SavingsAccount(const char* name, int id, double bal)
    : Account(name, id, bal), rate (6.0), interest(0) { }
  void payInterest()                                   // выплата раз в месяц
  { double pay = balance * rate / 12 / 100;
    balance += pay; interest += pay; }
```

```
  virtual void display()
  { cout.setf(ios::fixed,ios::floatfield); cout .precision(2);
    cout <<setw(6) << pin << setw(8) << interest << setw(12)
         << balance << " " << owner << endl; }
} ;
```

Здесь определяются объекты базового и производного классов. Кроме того, определяется указатель Account и используется операция dynamic_cast для его установки сначала на базовый объект, а затем на производный объект. При использовании операции dynamic_cast следует проверить, является ли указатель нулевым. Если да, то ответ на вопрос отрицательный. Если указатель не нулевой, ответ утвердительный. Объект, который обозначает указатель, может использоваться как объект класса, определенного в операции dynamic_cast.

```
Account a1 ( "Jones", 1122,5000) ;                    // создать объекты
SavingsAccount a2 ( "Smith",1133,3000);
Account *pa = dynamic_cast<Account *>(&a1);           // ok
if  (pa == 0)
    cout << "Null pointer\n";
else
    pa->display();                                    // Jones
pa = dynamic_cast<SavingsAccount *>(&a2);             // ok
if (pa == 0) cout << "Null pointer\n";
else
    pa->display();                                    // Smith
```

В этом примере ответы на вопрос не очень важны, поскольку указатель, используемый как цель присваивания, является базовым. Первое приведение возвращает указатель на объект a1, а второе приведение — на объект a2, который преобразуется в базовый указатель. Поскольку функция display() является полиморфной, она отображает данные в первом случае в базовом формате и во втором случае в производном формате. В этом фрагменте программы демонстрируется поведение операции dynamic_cast.

В следующем фрагменте программы базовый указатель снова используется как цель. Во-первых, он указывает на базовый объект, у которого запрашивается, может ли он выполнить операции Account. Ответ утвердительный, т. е. операция возвращает базовый указатель, ссылающийся на интересующий объект. Тем не менее сообщение display(), посылаемое через этот указатель, указывает на данные в формате производного класса, потому что функция является виртуальной. Объект принадлежит производному классу. Затем проверяется, может ли объект a1 выполнить обязанности объекта SavingsAccount. Ответ отрицательный, это объект базового класса и операция возвращает нуль.

```
pa = dynamic_cast<Account *>(&a2);                    // ok
if  (pa == 0)
    cout << "Null pointer\n";
else
    pa->display();                                    // Smith
pa = dynamic_cast<SavingsAccount *>(&a1);             // нулевой
if (pa == 0) cout << "Null pointer\n";
else
    pa->display();
```

Следующий фрагмент программы намного интереснее, поскольку в нем используется указатель производного класса. Сначала проверяется, может ли объект a1 выполнить обязанности производного класса. Как и в предыдущем случае, ответ отрицательный. Совершенно все равно, является ли целью присваивания

базовый указатель (как в предыдущем случае) или производный указатель (как в данном случае) — нуль всегда нуль. Затем проверяется, может ли объект a2 выполнить обязанности производного класса. Ответ, как и для самого первого фрагмента программы, положительный. В первом фрагменте результат преобразовывался в базовый указатель, следовательно, мог вызывать только методы Account и виртуальные методы производного класса. Здесь преобразование отсутствует. Целью присваивания является производный указатель. Он может осуществлять доступ к базовым функциям, виртуальным функциям и функциям, определенным в производном классе.

```
SavingsAccount *psa = dynamic_cast<SavingsAccount *>(&a1);     // 0
if  (psa == 0)
    cout << "Null pointer\n";                    // нулевой указатель
else
    psa->display();                              // без отображения
psa = dynamic_cast<SavingsAccount *>(&a2);       // ok
if  (psa == 0)
    cout << "Null pointer\n";
else
    {  psa->payInterest();                       // производный метод
       psa->display(); }                         // Smith
```

Понятно, что операция dynamic_cast представляет собой мощный метод проверки, может ли указанный объект выполнить требуемую операцию. Не все компиляторы поддерживают такую возможность. Те, которые поддерживают ее, могут не делать это по умолчанию. Для использования данной возможности необходимо установить флаги компиляции или опции для явной поддержки возможностей RTTI.

Подобно оператору new, C++ поддерживает другой метод проверки, будет ли успешным приведение типов: генерация исключительной ситуации. Если указатель не ссылается на объект класса, определенного в операции, то генерируется исключительная ситуация bad_cast. Это важно для ссылок. Указатели C++ могут как указывать, так и не указывать на объект, но ссылки C++ делают это всегда. Они не могут иметь нулевое значение. Для ссылок не нужно использовать dynamic_cast, чтобы показать, что действительно указывает на объект класса, определенного в приведении. Когда формальное утверждение оказывается неверным, соответствующим является генерация исключительной ситуации.

Операция typeid

Для определения того, к какому классу следует выполнить приведение базового указателя, используется операция typeid. С помощью операции typeid можно выполнить одно из двух: проверить имя класса параметра или уточнить, относится ли к данному классу объект, на который установлен указатель.

В отличие от операций приведения типа операция typeid работает с параметром-объектом, а не с параметром-указателем. Она возвращает ссылку на объект type_info библиотечного класса. Реализация этого вызова зависит от компилятора. Однако среди своих компонентов класса она всегда включает функцию-член name(). Эта функция возвращает символьный массив, т. е. снова зависит от реализации. Чаще всего это имя класса, к которому принадлежит параметр операции или имя класса с предшествующим зарезервированным словом class.

Некоторые реализации компилятора определяют конструкторы type_inf как закрытые. В таком случае программа C++ не может создать объект класса type_info. Вместо этого она должна послать сообщение в возвращаемом значении оператора typeid.

```
Account a1("Jones", 1122, 5000);          // создать объекты
SavingsAccount a2("Smith", 1133, 3000);
const char *c1 = typeid(a1).name();       // получить имя класса
const char *c2 = typeid(a2).name();
cout << c1 << endl;                       // выводит "class Account"
cout << c2 << endl;                       // выводит "class SavingsAccount"
```

Операция typeid не может быть выражением некоторого класса, а также именем класса, указанным как идентификатор (без кавычек). Это позволяет сравнивать результаты использования операции typeid с именем класса и с объектом. Если операция равенства возвращает true, то объект принадлежит к указанному классу.

```
if  (typeid(Account) == typeid(a1))          // true
    cout << "a1 is Account\n";
if  (typeid(SavingsAccount) == typeid(a2))   // true
    cout << "a2 is SavingsAccount\n" ;
if  (typeid(Account) == typeid(a2))          // false
    cout << "a2 is Account\n";
if  (typeid(SavingsAccount) == typeid(al))   // false
    cout << "a1 is SavingsAccount\n";
```

Во время работы с набором неоднородных объектов, обозначенных базовыми указателями, указатели, используемые как параметры в typeid, должны быть разыменованы.

```
pa = &a2;
if  (typeid(Account) == typeid(*pa))             // false
    cout << "pa points to Account\n";
if  (typeid(SavingsAccount) == typeid(*pa))      // true
    cout << "pa points to SavingsAccount\n"
```

Обратите внимание, что эти операции не сравнивают объекты. Для структур C++ операции сравнения не определены. В них не сравниваются указатели. В отличие от других специальных приведений оператор typeid возвращает объект, а не указатель. Он применяет перегруженную операцию сравнения к объекту type_inf, который возвращается операцией typeid.

Операция typeid — мощное средство, которым легко злоупотребить. Используйте ее как можно реже.

Итоги

Возможности, рассмотренные в данной главе, относительно новые. Не все компиляторы и реализации библиотек их поддерживают. Советуем вам использовать их с осторожностью.

Для выполнения исключительных ситуаций требуется затратить время и память. Это может оказаться важным для некоторых приложений. Важно правильно использовать исключительные ситуации для структуризации и упрощения передачи управления в приложении.

Может показаться, что исключительные ситуации, которые генерируют значения встроенных типов, не слишком полезны. Ведь программы обработки исключительных ситуаций не могут отличить значения одинакового типа, сгенерированные в разных местах исходной программы. Проектирование классов для передачи информации исключительной ситуации от места обнаружения ошибки к месту ее исправления очень полезная и интересная задача.

Многие подтверждают, что использование исключительных ситуаций упрощает передачу управления в приложении и позволяет проектировщику отделить основную обработку от запутанных исключительных случаев. Возможно, использование управляющих структур `if` и `switch` действительно приводит в замешательство — исходный текст программы становится трудным для понимания. Имейте в виду, что сложная структура программы отражает трудные для понимания задачи. Несомненным преимуществом использования стандартных управляющих структур для обработки различных случаев является то, что вся программа обработки находится в одном месте, она не разделена на части.

При использовании исключительных ситуаций программист, осуществляющий сопровождение, должен выполнять анализ решений, сделанных проектировщиком в отношении того, как разделить обработку на отдельные части. Эти решения часто сложны и имеют произвольный характер. Таким образом усложняется программа.

Представляется, что использование исключительных ситуаций оправдано в случае, когда программа проектируется таким образом, что ее часть, выполняющая исправление ошибок, не располагает информацией о той части программы, которая обнаруживает ошибку. Советуем вам использовать исключительные ситуации для передачи необходимой информации от одной части программы к другой. Однако помните, что необходимость передачи информации от одной части программы к другой возникает в том случае, когда принято решение о разделении обработки на две отдельные части. Пересмотр такого решения может исключить необходимость чрезмерной обработки исключительных ситуаций.

Убедитесь, что использование исключительных ситуаций C++ не слишком усложняет работу программиста, осуществляющего сопровождение. Обозначьте все исключительные ситуации, которые каждая функция генерирует или передает от своего сервера. Не трогайте исключительные ситуации, которые функция не сможет сгенерировать. Снабжайте комментариями обработку исключительной ситуации в исходном тексте программы и выделяйте документирование. Убедитесь, что каждая программа обработки исключительной ситуации проверена.

Программисты не имеют опыта использования операций приведения типов. Автор использует стандартные приведения C-типа, и кажется, что этого вполне достаточно. Конечно, можно неверно воспользоваться стандартными приведениями типа. Стандартные приведения типов позволяют осуществлять преобразования указателей, которые не имеют смысла, а операция `static_cast` не позволит это сделать. Но преобразования, не имеющие смысла, обычно не делаются. Если такое преобразование потребуется, операция `reinterpret_cast` позволит сделать это настолько же легко, насколько допускает стандартное приведение типа.

Тем не менее большинство экспертов соглашаются, что за счет использования операции приведения типов повышается надежность программ. Они хорошо заметны и привлекают внимание программиста, осуществляющего сопровождение. Рекомендуем вам использовать их на практике.

Глава 19

Полученные уроки

Темы данной главы

- ✔ C++ как традиционный язык программирования
- ✔ C++ как модульный язык
- ✔ C++ как объектно-ориентированный язык
- ✔ C++ и его конкуренты
- ✔ Итоги

Мы прошли длинный путь. Стоит признать, что эта книга создавалась долго. Иногда казалось, что последняя глава никогда не будет написана. Однако пришло время завершить свой тяжелый труд. Теперь можно оглянуться назад.

В этой главе не изучается новый синтаксис. Вместо этого попытаемся кратко обобщить основные характеристики этого великого, замечательного, запутанного языка C++. Теперь, когда вы усвоили весь предмет целиком и поняли, как разные компоненты состыковываются друг с другом, можно оценить, как много идей вложено в его проектирование и насколько осмотрительным нужно быть при его использовании.

В предыдущих главах были установлены некоторые ограничения в отношении того, что можно, а что нельзя обсуждать, поскольку некоторые вещи были недостаточно хорошо известны. Теперь это ограничение не имеет силы.

Язык C++ был создан как язык программной инженерии для построения больших компьютерных программ. Он преследовал несколько целей. С одной стороны, C++ старался быть языком программирования систем, от которых требуется высокая производительность: обеспечивая операции низкого уровня (например, сдвиги и побитовые логические операции) и поддерживая доступ к ресурсам компьютеров (например, регистры, изменчивые типы данных и арифметические операции над указателями). С другой стороны, C++ пытался облегчить разделение программ на независимые части, которые могли разрабатываться разными программистами, мало взаимодействующими друг с другом.

C++ является средством достижения нескольких конфликтующих целей. Он был спроектирован:

- Как удобочитаемый язык высокого уровня (агрегирование данных, передача управления, область действия имен)
- Как язык для острого и гибкого ума (уникальные по краткости записи операторов, лаконичные выражения)

- Для использования символьных строк и динамического управления памятью
- Для применения имеющихся библиотек (стандартов де-факто)

C++ как традиционный язык программирования

В отличие от многих языков высокого уровня язык C++ чувствителен к регистру. Подобно многим современным языкам высокого уровня, C++ не учитывает пробелы (за исключением двух или трех случаев). Комментарии приводятся в нем в конце строки, но сгруппированные в блоки комментарии не используются. Как и большинство других языков программирования, C++ обеспечивает основные встроенные типы данных в операциях над значениями этих типов. Встроенные типы данных C++ ограничены, применяются только целые значения и значения с плавающей запятой.

Встроенные типы данных C++

Целый тип является самым быстрым типом на любой платформе. Размер его составляет 16 бит на 16-битовой машине и 32 бита на 32-битовых машинах. Таким образом, вы можете столкнуться с проблемами переносимости, типичными для C++. Нет гарантии, что программа, которая выполняется на одной машине, даст подобные результаты на другой машине.

С целью повышения гибкости (т. е. для сохранения памяти) и добавления вычислительных возможностей (т. е. для расширения диапазонов, когда необходимо) для сложных вычислений C++ предусматривает модификаторы размера (`short`, `long`, `unsigned`) для более тонкого использования памяти. В C++ не стандартизованы размеры различных типов. Он требует, чтобы короткое значение не было длиннее целого значения, а также, чтобы длинное значение не было короче целого значения.

На современных машинах короткие значения всегда занимают 16 бит, а длинные значения — 32 бита. Программисты, которые стремятся к переносимости, избегают явного использования целых значений и вместо них используют либо короткие, либо длинные модификаторы. Программисты, которые стремятся к скорости, используют простые целые и избегают использования коротких и длинных модификаторов.

Использование значений без знака предполагает более точное применение памяти и является более спорным. С одной стороны, определение значения без знака указывает программисту, осуществляющему сопровождение, что значение по сути является положительным и не может быть отрицательным. Использование квалификатора без знака удваивает максимальное целое значение при данной архитектуре (для одинакового числа бит). С другой стороны, совокупность значений со знаком и без знака может привести к неверным результатам вычислений. Для того чтобы избежать ошибок, многие программисты не используют значения без знака.

Для упрощения программистам выбора C++ поддерживает значения по умолчанию. Если программист не укажет, со знаком число или без знака, то по умолчанию считается, что знак есть. Если программист не определит, является ли значение коротким целым, длинным целым или просто целым, значение по умолчанию представляет собой целое значение.

Поскольку вы стремитесь повысить производительность, C++ не проверяет результаты вычислений ни на потерю значимости, ни на переполнение. Все, что следует проверить в программе, должно быть изучено явно в Ее исходном тексте в определенное время. Если программа не желает тратить время на проверку допустимости результата, то в C++ не предусматривается каких-либо тестов по умолчанию или предупреждений.

Символы в C++ интерпретируются как особый вид целых значений. Их размер изменяется от одного байта на символ до двух байтов на символ (расширенный набор символов). Арифметические операции над символьными значениями допустимы в C++. Они широко используются, но могут создавать проблемы с переносимостью, когда различные машины применяют разные наборы символов.

Язык позволяет программисту определить как символы со знаком, так и без знака. Для типа по умолчанию стандарт отсутствует. На некоторых машинах применяются символы без знака, на других — со знаком. Предположите, что символ не может содержать отрицательное значение и использовать вместо символа целое, если допускаются отрицательные значения (например, код конца файла (end-of-file)).

Символьные литералы заключаются в одиночные кавычки. Не путайте их со строковыми литералами, заключаемыми в двойные кавычки. C++ не сохраняет длину строки в содержимом строки. Для указания конца строки используется код 0. Именно поэтому длина строкового литерала на единицу больше, чем количество символов в литерале.

Для типов с плавающей точкой C++ поддерживает три размера: `float`, `double` и `long double`. Их размер может быть 4, 8 и 10 байт, точности в диапазоне 7, 15 и 19 цифр. Характеристики эти зависят от машины. Константы в C++ с плавающей точкой всегда `double`, не `float` и не `long double`. В большинстве случаев это не важно. Когда необходимо объявить, что литерал, например, типа `float`, воспользуйтесь соответствующим суффиксом. C++ поддерживает как запись с фиксированной десятичной точкой, так и экспоненциальное представление чисел.

Логические типы имеют два значения: `true` и `false`. Они также представляются как малые целые. Размер логического значения типа `bool` один байт, а не один бит. C++ не упаковывает логические значения в один бит, поскольку для адресации отдельных битов в C++ требуются логические операции и сдвиги. При выборе компромисса между эффективностью использования памяти и эффективными затратами времени в C++ предпочтение отдается эффективности по времени, поскольку байт является наименьшей долей памяти, которую можно непосредственно адресовать.

Символические имена для значений литералов любых встроенных типов могут определяться с помощью директивы препроцессора `#define`. Препроцессор заменит каждое появление символического имени в исходной программе на литеральное значение. Поскольку эта процедура выполняется до того, как компилятор увидит исходный текст программы, часто ошибки в директивах препроцессора очень трудно отыскать. Лучше использовать модификатор `const`, потому что имена, определенные в модификаторе `const`, соответствуют правилам области действия (имена, заданные в директиве `#define`, являются глобальными).

Для каждого типа данных C++ поддерживает два производных типа данных: тип указателя и тип ссылки. Они содержат адрес значения, но синтаксис их отличается.

C++ допускает любые преобразования между числовыми значениями разных типов: значение одного типа может использоваться, когда ожидается значение другого типа. Логические значения и числовые значения также взаимозаменяемы — синтаксическая ошибка не выдается. Для числовых значений C++ является языком со слабым контролем типов.

Значения указателей (или ссылок) на разные типы невозможно преобразовать друг в друга. Для адресов C++ является языком со строгим контролем типов. Синтаксическая ошибка генерируется, когда указатели разных типов содержат одинаковый адрес.

Для преобразований указателей (и ссылок) может использоваться явное приведение типа, но за целостность результатов отвечает сам программист. Компилятор не выдает никаких синтаксических ошибок, если результаты не имеют разумного значения или не переносимы на компьютеры с другой архитектурой.

Выражения C++

Язык C++ включает следующие операции над числовыми значениями: операции со знаком, арифметические операции, операции отношений, операции равенства и логические операции. Отсутствует операция возведения в степень. Подобно большинству других языков программирования, он не содержит неявного умножения — в качестве явного оператора должен использоваться символ "звездочка".

C++ интерпретирует операции как выражения. Для достижения единообразия C++ интерпретирует присвоение и запятую как операции (хотя у них самый низкий приоритет). В результате ошибочные структуры могут быть приняты компилятором C++ как допустимый код.

Поскольку количество операций большое, в C++ используются двухсимвольные и даже одна трехсимвольная операция (знак операции вычисления выражения по условию). В C++ значение операции (и зарезервированного слова) часто повторно используется для различных целей и поэтому определяется с помощью контекста.

Поскольку размер встроенных типов данных зависит от машины, C++ позволяет программисту оценивать размер указанной переменной (указывая имя переменной) или размер любой переменной указанного типа (обозначая наименование типа). Логические операции, операции отношений и равенства возвращают булевы значения: `true` и `false`, но они могут свободно преобразовываться в числовые значения 1 (`true`) и 0 (`false`). Кроме того, когда ожидается булево значение, может использоваться любое числовое значение — синтаксическая ошибка не выдается. Значение 0 преобразуется в `false`, а любое другое значение интерпретируется как `true`. В этом случае компиляторы C++ иногда вынуждены принимать код, который является семантически неверным.

Другим источником ошибок является операция равенства, которая записывается как два последовательных знака равенства: отсутствие одного знака равенства не вызывает выдачу синтаксической ошибки, но коренным образом меняет смысл исходного текста программы.

Логические операции && и || имеют разный приоритет — операция && связывает более тесно, чем ||. Это позволяет избегать дополнительных скобок. Обе логические операции являются операциями ближнего действия. В составных логических выражениях оценивается первый операнд. Если результат операции известен из оценки первого оператора, то второй операнд не оценивается.

В C++ имеется множество уникальных операций, которые обеспечивают доступ к базовым представлениям информации в компьютерной памяти. Это побитовые логические операции, среди которых "включающее или", "исключающее или", "отрицание" (дополнение). Они обрабатывают каждый бит операнда по отдельности.

Побитовые сдвиги осуществляют сдвиг указанного набора битов влево или вправо. Когда набор сдвигается влево или положительное число сдвигается вправо, освободившиеся места дополняются нулями — эти операции являются переносимыми. Если отрицательное число сдвигается влево, результат зависит от реализации: освободившиеся места заполняются либо нулями (логический сдвиг), либо единицами (арифметический сдвиг). Названные операции не являются переносимыми.

Другой набор уникальных операций включает операции инкремента и декремента. Они эмулируют обработку, подобную языку ассемблера, создавая побочный эффект (увеличение или уменьшение на 1) для одного операнда единичного значения. Эти операторы могут быть префиксными или постфиксными. Префиксный оператор применяется первым, затем значение используется в остальных выражениях. Постфиксный оператор применяется после того, как значение используется в других выражениях.

В выражении в C++ определяется порядок оценки операторов. Однако он не обозначает порядок оценки операндов. Следовательно, в программе на C++

не следует полагаться на конкретный порядок оценки операндов в выражении. В частности, операнды с побочными эффектами (операции инкремента и декремента) становятся обычным источником проблем, связанных с обеспечением переносимости. Используйте операции инкремента и декремента в независимых выражениях, чтобы избежать проблем с переносимостью.

Еще одной уникальной операцией является операция вычисления значения выражения по условию. В зависимости от значения ее первого операнда она оценивает либо свой второй операнд (первый операнд — правда), либо третий операнд (второй операнд — ложь).

Другой набор уникальных операций C++ включает оператор арифметического присваивания и оператор-запятую. Эти операции помогают в написании лаконичной и выразительной программы на C++.

Бинарные операции C++ всегда применяются к операндам одинакового типа. Когда в исходной программе определяются операнды разных типов, C++ использует преобразования расширения. Операнд меньшего размера преобразуется к типу выражения с самым большим размером. В операторе присваивания значение, расположенное справа, преобразуется к типу значения слева.

Управляющая логика в C++

Как и в других языках, операторы C++ выполняются последовательно. Каждый оператор завершается точкой с запятой. Допускаются блоки (составные операторы). Они заключаются в скобки и могут содержать локальные переменные. После закрывающей скобки блока точка с запятой не указывается.

Составные операторы могут быть вложены, они являются телом функции или телом оператора управления. Область видимости локальных переменных, определенных во вложенном блоке, обозначается рамками блока.

C++ содержит стандартный набор управляющих структур. В операторе `if-else` не используется зарезервированное слово `then`. Когда результатом выражения оператора является значение любого типа, не равное нулю, выполняется ветвь `true`, а когда результат равен нулю (`false`), применяется ветвь `false`.

C++ реализует выполнение повторяющихся действий и поддерживает три формы операторов организации циклов: цикл `while` (допускает повторение до нуля), цикл `do-while` (выполняется хотя бы один раз) и цикл `for` (используется для фиксированного количества повторений).

Популярный стиль программирования на C++ состоит в объединении проверки непрерывного цикла с присваиванием. Будьте осторожны со скобками. Пропущенные скобки могут изменить смысл выражения, поскольку операция сравнения в C++ имеет более высокий приоритет, чем операция присваивания.

C++ не поддерживает неограниченные переходы. Оператор `goto` не может выйти за пределы своей области видимости и осуществить переход через определения переменных. Оператор `break` выполняет выход из цикла таким образом, что управление передается оператору, расположенному за циклом. Оператор `break` может использоваться во всех трех структурах организации циклов и обычно выполняется в условных операторах. Оператор `continue` пропускает оставшуюся часть тела цикла и возвращается к началу цикла для проверки условия последующих итераций.

Оператор выбора (`switch`) в языке C++ поддерживает в программе решение для множественного ветвления. Он обеспечивает переход на альтернативные ветви выполнения в зависимости от значения целочисленного выражения. (Нельзя использовать значения с плавающей точкой.) Ветвь по умолчанию выполняется, если не находится совпадение. В отличие от других языков оператор по умолчанию не является обязательным. Если он отсутствует и не найдено совпадение, задается следующий оператор. Для создания структур с несколькими ветвями операторы `break` должны использоваться в конце каждый ветви.

C++ как модульный язык

C++ поддерживает иерархии компоновочных блоков для данных и для операций программы. С точки зрения программной инженерии, к преимуществам разбиения больших проектов на модули относится разделение труда, упрощение задач программирования, повторно используемые и сопровождаемые программные компоненты и возможность изучения программы на разных уровнях: либо в общем плане (независимо от деталей), либо подробно (не учитывая вопросы высокого уровня).

За счет корректного использования подобных преимуществ повышается производительность как при разработке, так и при сопровождении программ. Кроме того, сокращается количество ошибок.

C++ поддерживает составные типы данных, определяемые программистами: массивы, структуры, объединения и перечисления. Их компоненты могут быть либо встроенных типов данных, либо других составных типов C++ (массивы, структуры и т. п.).

C++ поддерживает функции, определенные программистом. Иерархия функций повторяет иерархию действий реальных объектов, информация о которых сохраняется в программе. C++ использует стандартные библиотеки, которые обеспечивают выполнение большого количества разнообразных задач общего назначения. Библиотечные функции оптимизированы, хорошо протестированы и широко применимы.

Составные типы C++: массивы

Массивы C++ могут содержать элементы только одного типа. Самым большим ограничением массивов C++ является требование, что на момент выполнения компиляции размер массива должен быть известен. Если массив содержит больше элементов, чем необходимо, то напрасно расходуется память. Если массив включает меньше положенного количества элементов, то искажается информация в памяти.

Еще одним обычным источником ошибок при использовании массивов C++ является то, что индекс первого элемента всегда равен нулю. Изменить это невозможно. Следовательно, индекс последнего элемента всегда на единицу меньше диапазона массива. В C++ не поддерживается проверка индекса во время выполнения. Компилятор не будет это делать ни при каких обстоятельствах. C++ также не поддерживает проверку допустимости индекса во время выполнения.

Имейте в виду, что нежелательно расходовать напрасно время при любом обращении к массиву. Если требуется проверить допустимость значения индекса, можно написать собственную программу. Когда проверка допустимости индекса не выполняется, C++ предполагает, что так и надо. На машинах с большой памятью ошибки индекса могут не приводить к неверным результатам. Это серьезная проблема, которую трудно предотвратить и решить.

C++ позволяет программисту реализовать алгоритмы обработки массивов, используя для доступа к элементам массива либо индексы, либо указатели. Это основано на том факте, что при применении операции инкремента (или декремента) к указателю осуществляется приращение адреса не на единицу, а на длину элемента массива. Такая операция устанавливает указатель на следующий элемент массива. Использование указателей позволяет писать краткие и выразительные программы обработки массивов. Однако при использовании этого метода отсутствуют преимущества в плане производительности. Некоторые программисты полагают, что программы такого вида довольно трудно проверить.

C++ поддерживает массивы любой размерности. Они реализуются как одномерные массивы с развертыванием их по строкам. (Правый индекс изменяется быстрее остальных.) Одномерные и многомерные массивы C++ не поддерживают проверку допустимости индекса.

Текст в C++ представляется как массив символов. Эти массивы должны включать дополнительный элемент для размещения нулевого разделяющего значения, которое используется в качестве метки конца допустимых данных в массиве. Когда компилятор обрабатывает литеральные константы текста программы, он также добавляет завершающий нуль к символам строки. Следовательно, литералы содержат дополнительный элемент. Все библиотечные функции, обрабатывающие массивы символов, предполагают наличие завершающего нуля в конце допустимых данных. Когда библиотечные функции изменяют содержание массива, они добавляют завершающий нуль к концу допустимых данных, чтобы сохранить достоверное состояние строки.

C ++ не поддерживает ни присваивания, ни сравнения массивов. Для массивов произвольных типов именно программист отвечает за правильность выполнения этих операций. Для строк текста библиотечные функции используются для выполнения присваивания, сравнений, сцеплений и других стандартных операций.

Большинство библиотечных функций работает плохо, если строки частично перекрываются в памяти. При выполнении записи в символьный массив ни одна из библиотечных функций не проверяет наличие достаточного пространства. Если в памяти для строки мало места, то память компьютера искажается без выдачи соответствующего сообщения. А это угрожает целостности данных.

Составные типы C++: структуры, объединения, перечисления

Структуры C++ объединяют связанные компоненты. Программист решает, какие компоненты являются связанными, а какие нет.

Определение структуры является основой создания переменных типа структуры. Для каждого поля структуры программист указывает тип и имя поля. Область действия определения структуры ограничивается открывающей и закрывающей фигурной скобкой с последующей точкой с запятой.

Переменные типа структуры могут инициализироваться с помощью синтаксиса, сходного с синтаксисом инициализации массива.

Операция селектор-точка выбирает поля объектов структуры (как l-значение, так и r-значение). Когда указатель ссылается на переменную типа структуры, операция селектор-точка не работает. Вместо него должна использоваться операция селектор-стрелка.

C++ поддерживает присваивание переменных одного типа для структуры. Реализованы семантики значений: поля r-значения переменной типа структуры копируются побитно в поля l-значения переменных типа структуры.

Присваивание для переменных типа структуры с различным типом не разрешается, даже когда они одинакового состава и когда поля в определениях структур имеют одинаковые имена. Должно быть одинаковое имя типа. Обратите внимание на то, что использование возможности `typedef` не обеспечит одинаковое имя типа. Она только синоним для имени типа.

Также не допускаются присваивания между переменными типа структуры и числовыми переменными (или переменными указателями или ссылками). В отношении типов, определенных программистом, C++ является языком со строгим контролем типа. Такие присваивания помечаются компилятором как синтаксические ошибки.

C++ не поддерживает проведение различных операций в отношении структур. Реализовать операции со структурами можно в своих собственных программах.

Объединение является определением типа, который синтаксически подобен определению структуры. В списке между фигурными скобками области действия (с последующей точкой с запятой) можно перечислить несколько полей различных типов. Однако подобные поля существуют в памяти компьютера не одновременно (как в переменной типа структуры), а последовательно.

Такая схема позволяет программе сохранить пространство в памяти. Переменная типа union может содержать информацию одного из взаимоисключающих типов, указанных в определении типа union. Это способствует появлению ошибок, поскольку программист должен убедиться, что значение, извлеченное из переменной типа union, и значение, сохраненное ранее в этой переменной, совпадают, а само объединение не располагает средствами для сохранения информации о типе.

Если программа делает ошибку и извлекает значение другого типа, то сообщение об ошибке не выдается ни во время компиляции, ни во время выполнения. Бесполезный набор битов извлекается без каких-либо сообщений. Чтобы избежать подобных ошибок, переменные типа union могут использоваться как поля структур. К структуре можно добавить поле признака для сохранения информации о том, как было инициализировано значение поля union. Когда извлекается значение union, команда обращается к полю признака и действует соответствующим образом. Так обычно реализовывался полиморфизм, прежде чем были изобретены виртуальные функции.

Перечислимые типы определяют переменные, которые принимают значения из предварительно определенного набора символьных идентификаторов. Синтаксис определения перечислимого типа подобен синтаксису определения структуры. Набор символьных имен, разделенных запятыми, указывается в фигурных скобках (с последующей точкой с запятой). Это общераспространенный способ определения символьных констант в программе.

C++ не определяет операции над значениями перечислимого типа. Они реализуются как целое. Программа может использовать эту информацию, но мы не советуем это делать.

Функции C++ как средства обеспечения модульности

C++ позволяет скрыть внутреннюю сложность функционирования. Клиентская программа использует серверные функции как отдельные блоки программы. Это позволяет упростить код вызывающей программы в соответствии с ее назначением. Клиентская программа представляется функцией, вызывающей серверные функции, а не как последовательность деталей более низкого уровня из операций над данными.

В последнем случае, когда клиентская программа реализует обработку данных без вызова серверных функций, программист, осуществляющий сопровождение, должен понять смысл последовательности операторов. В случае использования серверных функций цель каждой операции выражается именем функции.

Функции C++ работают совместно друг с другом над достижением общей цели программы. Они обрабатывают общие данные. Значения данных могут устанавливаться одной функцией, а использоваться другой функцией. Обмен данными реализуется за счет использования глобальных переменных, параметров и возвращаемых значений. Связывание посредством глобальных переменных является неявным. Оно не сразу очевидно программисту, осуществляющему сопровождение, и, следовательно, должно использоваться как можно реже.

Связывание посредством параметров лучше, поскольку оно явное. Программисту, осуществляющему сопровождение (и программисту клиентской части), сразу же становится ясно, какие значения вовлекаются в поток данных между функцией и ее клиентскими функциями.

Связывание (количество параметров) должно уменьшаться до минимума посредством разделения обязанностей между функциями. Однако то, что составляет единое целое, не должно быть разделено между различными функциями. При разделении одного целого на части необходимо обменяться данными между функциями.

С помощью функции клиентская программа должна передать фактический параметр для каждого формального параметра в определении функции. Можно определить значения по умолчанию для параметров, чтобы они использовались, когда клиентская программа не обозначает значения фактических параметров.

В отношении передачи параметра C++ спроектирован как язык со строгим контролем типов. Количество аргументов должно совпадать с числом формальных параметров, а тип каждого аргумента с типом соответствующего формального параметра. Случаи, когда вы не придерживаетесь данного правила, помечаются компилятором как синтаксические ошибки.

Исключение из этого правила делается только для числовых типов. Если типы формального параметра и соответствующего фактического аргумента не совпадают, могут применяться преобразования типов. Малые аргументы (`enum`, `char`, `unsigned char`, `short`) повышаются до целых, короткие без знака становятся либо `int`, либо `int` без знака (в зависимости от архитектуры машины), а аргументы с плавающей точкой повышаются до типа `double`. Если после повышения тип фактического аргумента все еще не совпадает с типом формального параметра, то применяется преобразование. Любой числовой тип можно перевести в любой другой числовой тип, даже если это приводит к потере точности (например, из `double` в `integer`).

Повышения и преобразования не применяются к типам, определенным программистом, указателям и ссылкам. Они используются только для числовых типов.

C++ — язык с раздельной компиляцией. Для облегчения работы компилятора перед обработкой вызова функции компилятор должен знать ее интерфейс. Несмотря на то, что определение функции предшествует в исходном файле вызову функции, должен использоваться прототип с указанными типами параметров и возвращенного значения. Имена параметров в прототипе функции полезны, но не обязательны.

Функция C++ может определяться только один раз. Она может быть объявлена как прототип неоднократно. Если функция используется в нескольких файлах, она должна быть объявлена в каждом из них. Прототипы функций часто помещаются в заголовочные файлы `#include`.

Глобальная функция C++ определяется ее именем и последовательностью типов ее параметров. Когда функция определяется как член класса, имя класса также является частью определения функции. Эта комбинация (сигнатура функции) — имя класса (если имеется), имя функции и список типов параметров — должна быть уникальной. Следовательно, имя функции может быть перегружено. Функция с тем же именем, но с другим набором параметров, будет рассматриваться как другая функция. Возвращаемый тип не является частью сигнатуры функции.

Функция C++ может определяться как строковая функция. Вместо вызова функции компилятор генерирует для нее объектную программу и вставляет ее в клиентскую программу. Когда вызывается эта функция, не тратится время на контекстный переключатель. Для приложений, которые беспокоятся за скорость выполнения, это важно.

В C++ имеются только функции. Процедуры отсутствуют. Если для приложения требуется процедура, используйте функцию `void`.

Если функция возвращает значение, то C++ позволяет вызывающей программе игнорировать возвращаемое значение в вызове и использовать функцию как оператор. Многие библиотечные функции C++ содержат редко используемые возвращаемые значения. Однако игнорирование возвращаемых значений не является хорошей практикой.

Функции C++: передача параметров

В C++ параметры передаются по значению. В момент вызова в стеке выделяется память и локальные переменные, а значения аргументов (переменные, выражения или литералы) копируются в область памяти, выделенную для параметров. Эти значения используют функции. Когда выполнение функции завершается, выделенная память возвращается в стек.

При передаче параметров фактические аргументы передаются формальным параметрам. Измененные значения параметров обратно не передаются и фактические аргументы в области видимости клиента не изменяются.

Для побочных эффектов в клиентской программе C++ поддерживает передачу по указателю: вместо значения указанного типа как фактического аргумента передается указатель на значение. Указатели C++ являются переменными. Значение указателя копируется в формальный параметр. Когда указатель используется во время выполнения функции, он содержит адрес переменой в клиентском пространстве. Значение этой переменной при необходимости может изменяться через указатель.

Когда выполнение функции достигает закрывающей фигурной скобки, указатель вместе с другими формальными параметрами уничтожается. Следовательно, функция не может изменять значение указателя. Но это не проблема, поскольку изменять адрес нет необходимости. Цель передачи параметра по указателю — изменение переменной в клиентском пространстве, адрес которой передается как фактический параметр.

Передача параметров по указателю — сложный процесс. Программист должен скоординировать программу в трех точках:

1) запись указателя (*), которая используется в заголовке функции и в прототипе;

2) оператор разыменования (*), который применяется в теле функции;

3) адрес оператора (&), который используется при вызове функции.

Для упрощения передачи параметра C++ добавляет еще один режим передачи параметра, который поддерживает побочные эффекты в пространстве вызывающей программы. При передаче по ссылке координация различных точек в пространстве проще:

1) имя переменной с операторами, которое используется в заголовке функции,

2) операция ссылки (&), которая применяется в теле функции,

3) имя переменной без операторов, которая используется в вызове функции.

Поскольку этот режим передачи параметров поддерживает побочные эффекты, он может использоваться для выходных параметров встроенных типов.

Массивы в C++ передаются таким же образом независимо от того, изменяются ли его компоненты в теле функции или нет. Программу необходимо скоординировать в следующих трех направлениях:

1) имя массива с пустыми круглыми скобками в заголовке функции,

2) запись индекса (или имени массива без круглых скобок) в теле функции,

3)массива без круглых скобок в вызове функции.

Объекты структур C++ могут передаваться по значению, по указателю и по ссылке. По значению параметры передаются так: имя переменной без модификаторов используется во всех трех местах (в заголовке функции, в теле функции

и в вызове функции). Однако в нем не поддерживаются побочные эффекты в клиентском пространстве. Даже когда они не требуются, передача по значению может быть нежелательной при отправлении больших структур переменных. Копирование их может замедлить работу программы. Передача параметров по указателю сложна, но она поддерживает побочные эффекты в клиентском пространстве и в ней не требуется копирование данных даже для больших структур. Передача параметров структур по ссылке учитывает преимущества передачи по значению и передачи по указателю.

Такой режим передачи параметров используется как для ввода, так и для вывода параметров с типами, определенными пользователем. Чтобы сообщить программисту, осуществляющему сопровождение, какие параметры изменены функцией, а какие нет, проектировщик использует модификатор `const` для входных параметров, не модифицированных функцией. С помощью данного метода выражается явное намерение проектировщика в программе, а не в комментариях.

Область видимости и класс памяти в C++

Лексический контекст в C++ обычный. Объекты (переменные) определяются в начале (или в середине) области видимости, обозначенной фигурными скобками. Имена, определенные в области видимости, могут повторно использоваться в независимых областях видимости. Если имя повторно используется во вложенных областях видимости, то объект во внутренней области видимости скрывает объект во внешней области видимости.

Класс памяти (экстент) является промежутком во время выполнения, когда для объекта выделяется память, а его имя связывается с ячейкой памяти. Большинство переменных C++ принадлежат автоматическому классу памяти. Это те переменные, которые определяются локальными переменными в области видимости функции или блока. Память для автоматической переменной выделяется из системного стека, когда выполнение достигает открывающей фигурной скобки блока.

Автоматические переменные существуют (и на них можно сослаться по имени) от места объявления до закрывающей фигурной скобки. Автоматические переменные могут инициализироваться при объявлении. Если этого не происходит, значит, у них отсутствуют исходные значения по умолчанию. Когда они выделяются в стеке, то содержат случайный набор бит, оставшихся от предыдущего использования этой памяти.

Память, используемая для другого вызова функции или другого цикла в блоке, не всегда находится в заданном месте. Следовательно, автоматическая переменная не может передать данные между последовательными вызовами одной и той же функции.

Преимущество использования автоматических переменных состоит в том, что имя может повторно использоваться в различных функциях без какой-либо координации действий между разработчиками.

Глобальные (или внешние) переменные объявляются вне любой функции, в начале файла либо где-то в его границах. Область их действия распространяется от места объявления до конца файла. Память для глобальных переменных выделяется до начала выполнения программы (открывающая фигурная скобка в `main`) и возвращается, когда выполнение достигает закрывающей фигурной скобки`main`.

Глобальные переменные могут инициализироваться при объявлении. Если глобальная переменная не инициализирована, то по умолчанию ей присваивается исходное значение, равное нулю.

Глобальная переменная может быть повторно объявлена как локальная переменная в любом файле или в функции. Локальное имя скрывает глобальное имя в области действия блока, в котором определено локальное имя. Локальное имя использует другую область памяти и не оказывает влияния на память, выделенную для глобальной переменной.

Чтобы глобальные переменные стали видимыми в других файлах, используйте внешнее объявление. Это популярный метод обмена данными между функциями, реализованными в разных файлах.

Класс статической памяти C++ представляет собой компромисс при проектировании. Это зарезервированное слово, которое может использоваться в нескольких контекстах.

Когда глобальный объект определяется как `static`, память для него выделяется до начала выполнения `main` и освобождается при завершении программы. Но в отличие от обычной глобальной переменной статическую глобальную переменную невозможно сделать внешней в других файлах. Она видима только в одном файле.

Когда локальный объект определяется как статический, память для него также выделяется до начала выполнения `main` и освобождается по завершению программы. Статическая локальная переменная является закрытой в области действия — другие функции не могут осуществлять к ней доступ. Однако переменная и ее значение существуют даже вне области действия. Переменная становится доступной при последующем вызове функции.

Если функция инициализирует локальную статическую переменную, то она выполняется только один раз в начале программы. Когда функция завершается, статическая локальная переменная сохраняет свое значение, которое затем может использоваться в других вызовах функции. Это инструмент для обмена данными между различными вызовами одной и той же функции.

Класс динамической памяти передает программисту контроль за выделением и освобождением переменной. Существуют две опасности, которых должен избегать программист при использовании в программе динамического управления памятью: невозвращенная память, выделенная программе, и использование памяти, не заданной для программы.

Если память, выделенная программе, не возвращается, происходит утечка памяти. За счет этого уменьшается объем памяти для доступной программы, особенно когда она работает круглосуточно. Работа программы завершается аварийно или она выдает неверные результаты.

Использование памяти, не выделенной программе, не приводит к искажению данных в памяти. Программа завершается аварийно или выдает неверные результаты, пока не изменится использование памяти в компьютере.

C++ как объектно-ориентированный язык

C++ более совершенный язык по сравнению с C. Качество программы улучшается за счет использования таких возможностей, как комментарии в конце строки, гибкие определения переменных в середине области действия, символические константы для переменных и указателей, оператор области действия, приведения типа функции между типами, операторы `new` и `delete`, параметры по умолчанию, параметры, передаваемые по ссылке, перегрузка функций и операций, строковые функции, библиотека `iostream` объектов и операций ввода/вывода и др.

Однако основной вклад C++ в программную инженерию состоит в реализации объектно-ориентированных возможностей — классов, элементов данных и функций-членов, конструкторов и деструкторов, композиции классов и наследования, виртуальных функций, шаблонов и исключительных ситуаций.

Классы C++

Основная цель классов C++ — предоставить программисту инструмент для связывания зависимых данных и операций. Это исключает недостатки использования автономных глобальных функций для реализации операций в отношении значений типов данных, определенных программистом.

Данные определяются как элементы данных класса. Операции обозначаются как функции-члены класса. Синтаксис описания класса определяет элементы данных и функции-члены в области видимости класса, которая ограничивается открывающей и закрывающей фигурной скобками.

Язык C++ разрешает проектировщику класса определять права, которыми обладают другие части клиентской программы для доступа к компонентам класса. Общедоступные компоненты могут указываться из любого места, где можно описать объект класса. Защищенные компоненты обозначаются из функций-членов класса и из функций членов класса, для которых этот класс является базовым. На закрытые компоненты можно ссылаться только из функций-членов класса.

Следовательно, части программы, которые не принадлежат указанному классу, могут осуществлять доступ только к общедоступным компонентам класса. Это правило может работать не в полную силу при использовании "дружественных" классов и функций. Они обладают такими же правами доступа, что и функции-члены класса.

Обычно элементы данных определяются как закрытые, а функции-члены как общедоступные. Это не только связывает данные и операции с точки зрения программиста, осуществляющего сопровождение, но также скрывает реализацию данных от клиентской программы. С закрытыми данными и общедоступными операциями класс рассматривается клиентской программой как комбинация интерфейсов функций. Это поддерживает такие концепции современного программирования, как инкапсуляция данных и сокрытие информации.

Программирование с инкапсуляцией данных препятствует использованию в клиентской программе имен закрытых (или защищенных) элементов данных класса. Таким образом клиентская программа защищается от волновых эффектов при изменении реализации данных. Клиентская программа также защищена от волновых эффектов изменений в реализации функции-члена до тех пор, пока интерфейсы функций остаются без изменений.

Конечно, выгоды объектно-ориентированного программирования не обеспечиваются только благодаря использованию классов C++. Если функции-члены просто сохраняют значения элементов данных класса и извлекают их для использования в клиентской программе, то использование этих функций в клиентской программе не облегчает ни чтение, ни изменение программы. Проектировщик класса должен передать обязательства в серверный класс вместо извлечения клиентской программы. Для этого к классу, реализующему операции, существенные для достижения целей клиентской программы, добавляется функция-член.

Выражение клиентской программы в терминах вызовов функций-членов серверного класса делает программу понятной без пояснений, облегчает ее проектирование и сопровождение, уменьшает количество ошибок и улучшает ее качество.

Конструкторы, деструкторы и перегруженные операции

Конструкторы и деструкторы являются специальными функциями-членами, которые не могут иметь произвольных имен, определенных программистом. Имена получаются из имени класса, к которому они принадлежат. В свою очередь, программист клиентской части освобождается от обязанности явного вызова этой функции. Функции вызываются автоматически во время существования объекта класса.

Конструктор автоматически вызывается сразу же после создания нового объекта класса, в стеке или в динамически распределяемой области памяти. В классе может быть несколько конструкторов. Все они имеют одно имя класса, но у них — разные списки параметров. От количества и типа параметров, которые определены клиентской программой при создании объекта, зависит, какой конструктор вызывается.

В конструкторе проектировщик класса определяет способ элементов данных класса и способ выделения дополнительных ресурсов (например, память в динамически распределяемой области). Конструкторы не должны иметь возвращаемых типов и не могут возвращать значения. Если конструктор обменивается данными с вызывающей программой и передает информацию о возникновении проблемы при инициализации объекта, то он может сгенерировать исключительную ситуацию.

К важнейшим относятся конструкторы по умолчанию, конструктор копирования и конструкторы преобразования. Конструктор по умолчанию вызывается при создании объекта без передачи ему каких-либо параметров. Конструктор преобразования задается, когда указывается только один параметр (обычно с типом одного из элементов данных класса).

Конструктор копирования вызывается, когда другой объект того же класса используется для передачи данных инициализации объекту назначения. Он также вызывается, когда объект класса передается функции по значению.

Когда объекты класса динамически выделяют память в динамически распределяемой области, передача этих объектов по значению создает проблемы целостности данных. Программа завершается аварийно или ведет себя нестабильно (или выдает правильные результаты до тех пор, пока компьютер ведет себя стабильно при использовании памяти. Чтобы предотвратить возможные проблемы с целостностью, конструктор копирования используется для реализации семантики значений. Однако это замедляет работу программы. Лучше всего передать объекты класса по ссылке, а не по значению. Чтобы сделать передачу объектов класса по значению невозможной, конструктор копирования объявляется закрытым или защищенным.

Деструктор вызывается автоматически, перед тем как объект уничтожается. В деструкторе проектировщик класса определяет, каким образом ресурсы, выделенные объекту за время его существования, возвращаются системе.

Деструкторы не содержит возвращаемые типы, они не возвращают значения и у них не может быть параметров. Следовательно, перегрузка деструкторов невозможна.

Для использования конструкторов и деструкторов надо знать способ выполнения программы. В C++ (в отличие от других языков) отсутствует создание или уничтожение объекта с типом, определенным программистом. После создания объекта всегда вызывается конструктор, а перед уничтожением объекта вызывается деструктор.

Конструкторы и деструкторы для динамического управления памятью имеют большое значение. Вместо использования сложных и запутанных структур данных, состоящих из множества разнообразных компонентов, программист управляет узкой и хорошо определенной задачей по обработке только одного или двух компонентов. В результате уменьшается сложность задачи программиста и снижается вероятность появления ошибок.

Язык C++ одинаково интерпретирует объекты типов, определенных программистом, и переменные встроенных типов. В дополнение к перегрузке функций C++ поддерживает перегрузку операций. За счет этого клиентская программа может применять операции C++ к объектам класса.

Некоторые перегруженные операции предоставляют больше возможностей. Например, операцию, возвращающую элемент массива по индексу, можно использовать для принудительной проверки индекса и для предупреждения искажения данных в памяти, что является распространенной угрозой при использовании стандартных массивов C++. Оператор присваивания можно применять для избежания искажения информации в памяти и утечки памяти. Если семантика значений не нужна (один объект не должен присваиваться из другого объекта), оператор присваивания должен быть закрытым (или защищенным).

Композиция классов и наследование

Использование классов является наиболее эффективным методом. C++ поддерживает реализацию взаимосвязей среди объектов, используя указатели и ссылки. Элементы данных одного класса указывают на объекты другого типа.

C++ также поддерживает композицию классов, в которой объекты одного класса используются как элементы данных другого класса. Другой важной связью между классами, которую можно реализовать в C++, является наследование класса. При наследовании один класс используется как базовый класс, а второй класс применяется как класс, порожденный из базового класса.

При композиции класса C++ объект создается поэтапно во время выполнения. Сначала создаются компонентные объекты, а затем контейнерный объект. Поскольку создание объекта всегда сопровождается вызовом функции конструктора, задание составных объектов часто становится длинной последовательностью вызовов функций, которая может повлиять на производительность программы. Другая проблема состоит в том, что создание объекта приводит к вызовам конструктором других конструкторов, которые не реализованы классом, а это вызывает синтаксическую ошибку.

C++ предоставляет список инициализации, решающий обе эти проблемы. Синтаксис его необычен, но для бесхитростного подхода результат превосходный. Он определяет имена элементов данных, конструкторы которых, не определяемые по умолчанию, должны вызываться вместо конструкторов по умолчанию, и задает параметры для вызова этих конструкторов. Важно освоить этот метод инициализации объекта и широко использовать его.

Наследование в C++ позволяет программисту связать два класса концептуальной связью. В результате все, что определяется для базового класса, задается и для производного класса. Производный класс может определить некоторые дополнительные элементы данных и функции-члены, а также переопределить ряд функций-членов, используя то же имя и заменяя тело функции на более подходящее содержание.

С точки зрения клиентской программы, объект производного класса является комбинацией возможностей, определенных в базовом и в производном классах. Использование наследования — хороший способ повторного использования ПО. В этом случае возможности, определенные в базовом классе, используются в производном классе без написания какой-либо программы.

В результате элементы данных объекта производного класса определяются в базовом и в производном классах. Он также содержит функции-члены, определенные в базовом и в производном классах. Клиентская программа имеет доступ ко всем общедоступным компонентам (элементам данных и функциям-членам) объекта производного класса: как наследованным из базового класса, так и добавленным в производный класс. Программа производного класса имеет доступ к общедоступным и защищенным компонентам базового класса (как к элементам данных, так и к функциям-членам), но не к закрытым компонентам базового класса.

Синтаксис наследования повторно использует зарезервированные слова `public`, `protected`, `private` (и оператор двоеточия), но с другим значением. В открытом режиме наследования общедоступные, защищенные и закрытые элементы данных и функции-члены, определенные в базовом классе, остаются общедоступными, защищенными и закрытыми в объекте производного класса. В защищенном наследовании общедоступные базовые компоненты становятся защищенными в объекте производного класса и недоступны клиентской программе. В закрытом наследовании общедоступные и защищенные базовые компоненты являются закрытыми в объекте производного класса и недоступны для последующего порождения.

Это сложная система изменения прав доступа. Рекомендуем вам использовать открытое наследование, которое поддерживает естественные связи между классами. Значит, объект производного класса содержит все элементы данных и функции-члены производного класса.

При создании экземпляра для объекта производного класса сначала создается базовая часть. Вызывается конструктор базового класса. Отсутствие конструктора по умолчанию может вызвать синтаксические проблемы. C++ предоставляет список инициализации, который решает подобную проблему. Синтаксис его подобен списку инициализации для композиции классов, но вместо имен составляющих элементов данных используется имя базового класса. Освойте подобный метод инициализации объекта и используйте его.

Когда сообщение отправляется объекту производного класса, компилятор осуществляет поиск совпадающего определения производного класса. Если совпадения не обнаружено, то компилятор ищет базовый класс (или основу базового класса и т. д.). Если совпадение найдено здесь, то генерируется вызов соответствующей функции базового метода. Если совпадение не найдено, формируется синтаксическая ошибка. Если имя функции находится в производном классе, поиск прекращается. Если параметр совпадает со списком параметров, генерируется вызов функции метода производного класса. Если параметры не совпадают, поиск не возобновляется. Это синтаксическая ошибка даже в том случае, когда базовый класс содержит функцию с тем же именем, для которого совпадают параметры.

Следовательно, метод производного класса скрывает метод базового класса с тем же самым именем, независимо от сигнатур этих двух методов. Перегрузки функций в различных областях видимости с различными сигнатурами нет. Это часто является источником неочевидных ошибок, когда функция, которая вызывается фактически, отличается от функции, которую, как предполагает программист клиентской части, нужно вызвать.

Поскольку объект открыто порожденного класса имеет все свойства объекта базового класса, производный объект может использоваться в любом месте, где ожидается объект базового класса,— в присваивании, передаче параметра и манипуляции указателями. Преобразование из объектов производного класса в объекты базового класса является безопасным. Его можно выполнять всегда. При этом не требуется явное приведение типов.

Преобразование объекта базового класса в объект производного класса не является безопасным. Оно выполняется в случае, если имеется оператор явного присваивания или конструктор преобразования.

Преобразование указателя (ссылки) базового класса в указатель (ссылку) производного класса не является безопасным. Его нельзя выполнять неявно. Попытка сделать это помечается как синтаксическая ошибка. Если базовый указатель ссылается на объект производного класса, то преобразование в указатель производного класса имеет смысл. Чтобы убедить компилятор принять такое решение, следует воспользоваться явным приведением типов. Если программист совершает ошибку (базовый указатель не ссылается на объект производного класса), то возникает ошибка в период выполнения.

Виртуальные функции и абстрактные классы

Виртуальные функции расширяют концепцию сокрытия базовых функций в объектах производного класса. Эти функции используются, когда программа обрабатывает неоднородные объекты, принадлежащие к семейству подобных классов.

Каждый класс определяет функцию (используется то же самое имя, например `update`, и одинаковая сигнатура), которая выполняет сходную операцию (`update`) над объектами этих классов, но делает это по-разному для объектов разных классов (например, `SavingsAccount` и `CheckingAccount`).

Цель проектирования заключается в достижении полиморфного эффекта. То есть вы должны отправить сообщение update() каждому объекту в списке и вызвать либо сообщение update() класса SavingsAccount, либо сообщение update() класса CheckingAccount. При этом схожие классы порождаются из одного класса (например, Account).

Необходимо принять во внимание несколько ограничений. Порождение должно быть открытым. Базовый класс должен иметь метод с тем же самым именем и той же сигнатурой (например, update), что и производные классы. Этот метод должен быть определен как виртуальный. Неоднородный набор объектов требуется реализовать как список указателей базового класса, которые ссылаются на динамически распределенные объекты производных классов.

При соблюдении ограничений сообщение, отправленное базовому указателю, не вызывает метод базового класса, а задает метод производного класса, к которому принадлежит объект. Например, если базовый указатель ссылается на объект SavingsAccount, он вызовет метод update() из класса SavingsAccount. Если он указывает на объект CheckingAccount, он вызовет метод update() из класса CheckingAccount.

Многие программисты на C++ используют эту технологию. Такой метод красив и элегантен. Однако он не очень важен, поскольку обработка неоднородных списков не относится к наиболее распространенным задачам программирования.

Часто у базовых объектов такого семейства классов отсутствует задание в приложении (например, приложение обрабатывает сберегательные счета и счета до востребования, но не занимается неопознанными счетами). Последующее расширение данного метода преобразует базовый класс в абстрактный класс.

В абстрактном классе виртуальная функция определяется как чисто виртуальная функция, т. е. как функция, которая не имеет реализации. Вместо этого ее прототипу в спецификациях класса "присваивается" нулевое значение. Таков синтаксис, который определяет виртуальную функцию как чистую функцию, а базовый класс как абстрактный класс. Приложение не создает объекты абстрактного класса. Если программист задает объект абстрактного класса по ошибке, он помечается как синтаксическая ошибка.

Шаблоны

Шаблоны представляют собой еще одно средство для повторного использования ПО. Если приложению требуется контейнерный класс, который содержит компоненты разных типов, советуем вам не повторять процесс проектирования для каждого составляющего типа — эти классы почти идентичны.

Шаблоны C++ позволяют программисту создавать класс, в котором тип компонентов определяется как параметр класса. Когда задается экземпляр объекта этого типа, клиентская программа подставляет имя фактического типа, а компилятор генерирует код объекта класса, в котором каждый экземпляр параметра класса заменяется именем фактического типа, указанного клиентской программой.

Синтаксис шаблона сложен. Он включает использование списков параметров в угловых скобках. Синтаксис создания экземпляра объекта также сложен. Клиентская программа определяет фактические типы, также используя угловые скобки. Реализация функций-членов требует угловых скобок для списков формальных параметров и для имен классов в операторе области видимости.

Не совсем понятно, до какой степени отдельные приложения могут выиграть от использования шаблонов. Это относительно новая возможность, которая поддерживается еще не всеми компиляторами, и опыт ее использования ограничен. Однако стандартная библиотека шаблонов C++ (STL) использует шаблоны для реализации таких структур данных, как списки, очереди, векторы, хеш-таблицы и т. д. Данные классы хорошо спроектированы и оптимизированы. Несомненно, использовать их в индивидуальных приложениях полезно. Именно поэтому важно понимать синтаксис создания экземпляра и использования объектов-шаблонов.

Исключительные ситуации

Исключительные ситуации также представляют собой новую возможность C++, созданную для поддержки раздельной обработки основного алгоритма и исключительных ситуаций. Такое распутывание запутанной программы уменьшает сложность всей программы в целом.

Когда генерируется исключительная ситуация, создается объект встроенного или определенного программистом типа. Он отправляется от места возникновения исключительного состояния к месту, где исключительная ситуация должна будет обрабатываться. Это значение (или объект) передает информацию, которая полезна для обработки исключительной ситуации (например, строку с сообщением об ошибке).

Синтаксис использования исключительных ситуаций требует от программиста написания операторов обозначения исключительных ситуаций, генерации исключительных ситуаций и отслеживания их.

При обозначении исключительных ситуаций программист использует зарезервированное слово `throw` между заголовком функции и телом функции. За ним следует список типов, которые эта функция может сгенерировать (в круглых скобках, разделенные запятыми). Если определяется прототип функции, то зарезервированное слово `throw` и список вставляются между закрывающей скобкой списка параметров и последней точкой с запятой.

При генерации исключительной ситуации программист использует зарезервированное слово `throw` с одним параметром (значение встроенного типа или объект типа, определенного пользователем). Данный оператор применяется внутри условного оператора. Проверяется некоторое условие, и если его результат будет `true`, то генерируется исключительная ситуация для уведомления об этом программы обработки исключительных ситуаций.

Если оператор `throw` генерирует исключительную ситуацию класса, определенного программистом, и объекту должны быть переданы данные, то класс исключительной ситуации определит структуру, которая принимает необходимые значения и инициализирует элементы данных объекта. Если объекту не нужны никакие данные, вызывается конструктор по умолчанию. В отличие от всех остальных случаев создания объекта с помощью конструктора по умолчанию здесь после имени объекта должны стоять пустые круглые скобки.

Отслеживая исключительные ситуации, следует использовать сложные синтаксические конструкции — блок `try` с оператором, который сможет сгенерировать исключительную ситуацию, и ряд блоков-"ловушек", которые следуют после блока `try`. Каждый блок-"ловушка" проектируется для обработки исключительной ситуации только одного типа.

Блок `try` является безымянным блоком, которому предшествует зарезервированное слово `try`. Блок-"ловушка" — блок, перед которым располагается зарезервированное слово `catch` и список параметров только с одним параметром. Тип параметра `catch` является таким типом (встроенным или определенным программистом), для обработки которого спроектирован этот блок-"ловушка".

Если операторы в блоке `try` не генерируют исключительную ситуацию, выполнение блока `try` завершается. Если один из операторов в блоке `try` генерирует исключительную ситуацию, часть программы от этого места до конца блока `try` пропускается. Она не будет выполняться, даже если исключительная ситуация отслежена и обработана.

Затем блоки-"ловушки" последовательно проверяются на совпадение типа в списке параметров и типа значения, сгенерированного исключительной ситуацией. Если значение не совпадает, проверяется следующий блок-"ловушка". Если значение совпадает с типом параметра, блок-"ловушка" выполняется и делает то, что положено для обработки этой исключительной ситуации. Затем остальная часть блока-"ловушки" пропускается. Выполняется оператор, расположенный после последнего блока-"ловушки".

Если совпадение не найдено, то операторы, расположенные после последовательности try-catch, пропускаются, а выполнение функции завершается. Перед завершением она генерирует исключительную ситуацию, которая не была отслежена, и поиск продолжается в программе, вызвавшей эту функцию. Если блок-"ловушка", который обрабатывает исключительную ситуацию, обнаруживается, выполнение программы продолжается. Если нет, то поиск распространяется на main(), а выполнение программы завершается.

В результате метод приводит к сложным образцам кодирования, потому что программист может поместить последовательность try-catch в любую функцию и в любой комбинации. Часто сложно понять, в каком месте обрабатывается исключительная ситуация и как программа продолжает выполнение. Действительно ли использование исключительных ситуаций упрощает обработку основной линии алгоритма и облегчает понимание обработки исключительных ситуаций?

Преимущество этого метода состоит в том, что место обнаружения ошибки может быть отделено от места исправления ошибки. Всю необходимую информацию вы можете отправить от места обнаружения к месту исправления внутри объекта исключительной ситуации, сгенерированной оператором throw. Другое преимущество заключается в том, что деструкторы вызываются для всех объектов, удаленных из стека в процессе завершения функций, не содержащих соответствующий блок-"ловушку". Следовательно, если программа исправляет ошибку и продолжает выполнение, то она не пострадает от утечки памяти.

C++ и его конкуренты

C++ конкурирует со всеми языками программирования, от FORTRAN и COBOL до PL/I и Ada.

C++ и старшие языки

Конечно, даже у старших языков есть свои сильные стороны. К примеру, FORTRAN лучше всех подходит для реализации научных алгоритмов. Его библиотеки серьезно облегчают работу программистов, решающих научные задачи.

COBOL и PL/I превосходят C++, когда требуется гибкое форматирование вывода. Программистам, использующим C++, придется изрядно потрудиться, чтобы достичь подобного результата, особенно при использовании библиотеки iostream. Если применяются старые стандартные библиотеки, часть программы, выполняющая форматирование вывода, может быть короче. Все же это довольно сложно и порождает ошибки.

Ada включает такие возможности, как параллельное программирование и пакеты для реализации простых объектов. В этом случае программист может реализовать основные объектно-ориентированные структуры.

Однако ни один из этих языков не поддерживает композицию, наследование и виртуальные функции. Объектно-ориентированные возможности дают C++ превосходство над этими языками при написании крупных приложений

C++ и Visual Basic

Одним из интересных конкурентов C++, который становится все более популярным, является Visual Basic. За последние годы Visual Basic стал мощным и гибким средством программирования.

Возможности форматирования вывода, которые поддерживает Visual Basic и которые позволяют ему конкурировать с COBOL и PL/I, намного лучше возможностей, поддерживаемых C++. Visual Basic предоставляет программистам быстрый и легкий доступ к построению интерактивного ввода и вывода с использованием графических интерфейсов пользователя (Graphical User Interfaces — GUIs). Чтобы достичь подобных эффектов, программистам на C++ потребуется

изучить библиотеку для создания оконного интерфейса, а все библиотеки, имеющиеся сегодня на рынке, сложны для понимания и использования.

Кроме того, Visual Basic поддерживает некоторые объектно-ориентированные возможности, хотя его объектно-ориентрованные возможности не позволяют ему конкурировать с возможностями C++. Обучение на Visual Basic легче, чем на C++. Изучая Visual Basic, можно намного быстрее создавать осмысленные приложения, чем на C++.

Однако объектно-ориентированные возможности C++ интегрированы в язык намного лучше, чем в Visual Basic, где они являются красивым, но чужеродным дополнением.

C++ и C

Еще один язык, с которым C++ должен конкурировать сегодня, это его собственный предшественник — язык программирования C.

Языку C отдают предпочтение программисты, разрабатывающие системы реального времени и встроенные системы. Программисты опасаются таких возможностей C++, как виртуальные функции, шаблоны, библиотека `iostream` и исключительные ситуации. Они убеждены, что все это влияет на производительность и увеличивает размер выполняемой программы. Для систем реального времени и встроенных систем как производительность, так и размер исполняемых файлов имеют первостепенное значение. Именно поэтому C++ для такого вида приложений используется очень осторожно.

Думается, что это заблуждение. C является языком, разработанным для живого и острого ума, и это ведет к использованию кратких и выразительных образцов программирования. Такие образцы программирования иногда совершенно запутывают читателя, особенно когда программист пытается уменьшить до минимума размер объектной программы.

В результате организации, применяющие язык C для разработки систем реального времени и встроенных систем, втягиваются в реализацию сложных схем использования памяти, сложных структур глобальных данных с динамическим управлением памятью и запутанными соглашениями о вызовах. Структура таких систем настолько сложна, что для них невозможно обеспечить нужную документацию, учитывая жесткие планы выпуска версий. На обучение новых сотрудников тратится много времени, что отрицательно сказывается на производительности имеющегося персонала. Неправильные представления и неполное понимание решений, принятых для других частей системы, ведет к ошибкам проектирования и сопровождения, исправление которых обходится дорого.

Нельзя быть полностью уверенным в том, что все описываемое является универсальным. Однако несколько компаний, которые сопротивлялись переходу с C на C++, сталкивались с подобной ситуацией.

Очевидно, что использование классов C++ с закрытыми элементами данных и открытыми функциями-членами приводит к лучшему разбиению программы на модули, не оказывая при этом влияния на ее производительность и на размер объектной программы. Надлежащее использование конструкторов и деструкторов исключает ошибки управления памятью и облегчает их поиск. Кроме того, ограниченное использование виртуальных функций не оказывает влияния на размер и производительность программы.

Полиморфные алгоритмы можно реализовать на любом языке. При их реализации, например, на языке C, выделяется дополнительная память для сохранения информации о происхождении объекта, обрабатываемого программой. Программа тратит некоторое время на выяснение того, как интерпретировать этот объект. Использование C++ инкапсулирует данные сложности для программиста и не расходует напрасно дополнительные ресурсы.

К сожалению, многие компиляторы C++ выдают раздутый объектный код независимо от того, используют ли программисты шаблоны, библиотеки `iostream`,

исключительные ситуации или идентификацию при выполнении. При ограниченном использовании расширенных возможностей C++ поставщики компиляторов должны сотрудничать, чтобы программисты могли выбрать, что должно быть включено в каждую версию. Со временем все программисты, использующие C, перейдут на C++, даже если не все возможности C++ будут использоваться в каждом приложении.

C++ и Java

Сегодня язык Java наиболее опасный конкурент C++ . Java является дальним родственником C++. Подобно C++, он был создан как расширенное множество C. Большая часть объектно-ориентированного синтаксиса Java позаимствована из C++ (с некоторыми изменениями).

Java был разработан для поддержки классов с элементами данных, функциями-членами, конструкторами, виртуальными функциями и исключительными ситуациями. Программа на языке Java — это набор взаимодействующих объектов, предоставляющих сервисы друг другу. Java поддерживает композицию, наследование классов и полиморфизм. В Java поощряется повторное использование классов, определенных программистом, и повторное применение библиотечных классов для графического визуального пользовательского интерфейса.

В отличие от C++ при разработке Java не стояла задача обеспечения обратной совместимости с C. В результате язык Java не содержит значительного количества возможностей, способствующих появлению ошибок, которые C++ унаследовал от C. Кроме того, Java не включает многие возможности C++, которые либо приводят к увеличению размера объектной программы, либо поощряют использование методов программной инженерии более низкого уровня.

Вы можете услышать о том, что в Java отсутствуют указатели, следовательно, исходная программа на Java намного проще программы на C++. Программисты, которые отмечают это, просто не знают, о чем они ведут речь. В Java есть указатели. Те, кто не верит, могут выполнить простую программу и увидеть сообщение "Исключительная ситуация — нулевой указатель" ("Null pointer exception"), которое появляется на экране перед аварийным прекращением программы.

Java содержит явную операцию `new`, подобную C++. Однако в Java отсутствует явная операция `delete`. Вместо этого Java использует "сборку мусора". Это главное отличие от идеологии C/C++. Чем меньше времени тратится на отладку управления памятью, тем больше его остается для других алгоритмов.

Использование "сборщика мусора" вместе с интерпретацией при выполнении приводит к тому, что программы на Java работают намного медленнее, чем на C++. Несколько лет назад никто не стал бы использовать подобный язык. Но сегодня производительность уже не является самым важным критерием при оценке языков программирования. Переносимость, надежность и простота намного важнее (при условии, что язык поддерживает современный объектно-ориентированный подход, основанный на программной инженерии).

Приложения Java являются переносимыми. Все типы данных Java имеют стандартные размеры на всех машинах. Целое всегда имеет длину в 4 байта. Это не обязательно самый быстрый тип на рассматриваемой платформе, но программа выполняется одинаковым способом на всех машинах. Нельзя выбирать между типами данных со знаками и без знака. У числовых типов всегда есть знаки, а у логических и символьных типов их нет. Идентификаторы Java могут быть любой длины.

Java — "правильный" язык. Не разрешаются неявные приведения числовых типов. Явные приведения типов между числовыми типами допускаются, но не между числовыми типами и булевыми типами. В отличие от C/C++ реляционные и логические операции возвращают булевы значения `true` и `false`, а не 1 или 0. Неверное использование операции сравнения `==` как оператора присваивания `=` помечается как синтаксическая ошибка.

Java прост. В нем отсутствуют многие возможности C++, порождающие ошибки: перегрузка операторных функций, групповые шаблоны, множественное наследование, передача параметров по указателю, операции над незащищенными массивами, связанные с указателями, "дружественные" классы и функции, глобальные функции и переменные с именами, действующими в рамках проекта. (Каждая функция Java, включая `main`, должна быть членом некоторого класса.)

В Java отсутствует препроцессор с его макросами, включаемыми файлами и условной компиляцией. Не требуются прототипы функций — главный источник проблем в C++. Компилятору Java известно, где размещаются библиотеки.

Java реализует объектно-ориентированные возможности, которые отсутствуют в C++. Все функции являются виртуальными функциями по умолчанию. Наследование поддерживает как сокрытие базовых функций в производных классах, так и перегрузку, если различаются сигнатуры. Классы могут объединяться в пакеты. В Java имеется конструктор интерфейса, который лучше управляет использованием объектов одного класса, когда ожидаются объекты другого класса, чем наследование. В Java поддерживаются потоки, что помогает реализовать параллельное выполнение.

Вместо генерации машинно-зависимого объектного кода компилятор Java генерирует так называемый байт-код, для выполнения которого требуется интерпретатор. Это выглядит как помеха, не так ли? Тем не менее это означает, что байт-код Java может выполняться на любой платформе, на которой установлен интерпретатор Java. Программа, скомпилированная в среде Solaris, может выполняться на любой UNIX, PC или Mac машине без каких-либо изменений. Программисты на C++ не могут даже мечтать о таком уровне переносимости.

Именно поэтому Java стал языком Интернета. Байт-код, сгенерированный на одной платформе, может загружаться с другой платформы и выполняться, причем даже не запрашивая платформу сервера. Программы на Java используются и в неоднородных, и в однородных сетях.

Java поставляется с огромной библиотекой классов GUI. Действительно, изучение этих классов — задача не из приятных. Однако начинающий программист может создать имеющее смысл приложение на Java так же, как и написать приложение на Visual Basic, и намного быстрее, чем создать простое приложение на C++.

В Интернете Java абсолютный победитель. В работе с базами данных предпочтение отдается языку C++.

Итоги

Мы подошли к концу. Хотелось быть честным относительно преимуществ и недостатков C++. Замечательные возможности назывались замечательными. О возможностях, имеющих недостатки, говорилось, что они имеют недостатки. Об опасных возможностях предупреждалось, что они опасны.

Однако хотелось показать, что эффективное использование C++ требует изменения самого подхода к программированию. Необходимо продумать распределение обязанностей между различными частями программы и передачу их в серверные классы. Кроме того, необходимо оценить проектные решения, которые были бы понятны сопровождающему программисту непосредственно в коде, без комментариев. Попытайтесь использовать объекты некоторого класса там, где ожидаются объекты другого класса.

С помощью таких принципов программной инженерии можно создавать надежные, переносимые, повторно используемые и сопровождаемые приложения на C++. Делайте это с удовольствием, поскольку программирование на языке C++ должно приносить радость.

Желаю удачи!