ПРОГРАММИРОВАНИЕ



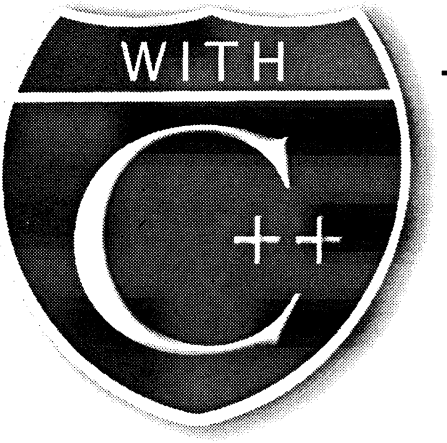
4-е издание

УОЛТЕР САВИЧ

Addison-Wesley
bhv
ПИТЕР

WALTER SAVITCH

PROBLEM SOLVING



THE OBJECT OF PROGRAMMING

Fourth Edition

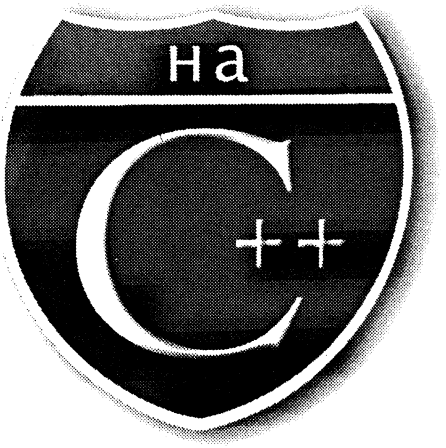


Addison-Wesley

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

УОЛТЕР САВИЧ

ПРОГРАММИРОВАНИЕ



4-е издание

ПИТЕР® bhv®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2004

ББК 32.973-018.1
УДК 681.3.06
С13

С13 **Программирование на C++. 4-е изд.** / У. Савич. — СПб.: Питер; Киев: Издательская группа BHV, 2004. — 781 с.: ил.

ISBN 5-94723-582-X

Книга содержит исчерпывающую информацию о языке программирования C++. Помимо «стандартных» тем, таких как объявление переменных, операторы выбора, циклы, массивы, функции и др., подробно рассматривается также работа с векторами, динамические многомерные массивы, обработка исключений, указатели и перегрузка операторов. Примеры и задания для самостоятельной работы, содержащиеся в каждой главе, помогут читателю закрепить изученный теоретический материал.

Книга рассчитана на студентов и начинающих программистов, которые хотят изучить тонкости программирования на языке C++.

ББК 32.973-018.1
УДК 681.3.06

Права на издание получены по соглашению с Addison-Wesley Longman.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 0321113470 (англ.)
ISBN 5-94723-582-X

© 2003 by Pearson Education, Inc.
© Перевод на русский язык ЗАО Издательский дом «Питер», 2004
© Издание на русском языке, оформление ЗАО Издательский дом «Питер», 2004

Краткое содержание

Предисловие 17
Глава 1. Основы информатики и программирования на языке C++ 19
Глава 2. Основные понятия C++ 50
Глава 3. Процедурная абстракция и функции 106
Глава 4. Функции для разных подзадач 156
Глава 5. Потоки ввода-вывода 192
Глава 6. Определение классов 254
Глава 7. Поток управления программы 310
Глава 8. Дружественные функции и перегрузка операторов 370
Глава 9. Раздельная компиляция и пространства имен 416
Глава 10. Массивы 447
Глава 11. Строки и векторы 515
Глава 12. Указатели и динамические массивы 558
Глава 13. Рекурсия 598
Глава 14. Шаблоны 633
Глава 15. Указатели и связные списки 656
Глава 16. Наследование и полиморфизм 683
Глава 17. Обработка исключений 725
Приложения 749
Алфавитный указатель 768

Содержание

Предисловие **17**
Последовательность изучения курса 17
Порядок изложения материала в главах 18
Где получить дополнительные материалы 18
От издательства 18

Глава 1. Основы информатики и программирования на языке C++ **19**
1.1. Компьютерные системы 19
Аппаратное обеспечение 19
Программное обеспечение 24
Высокоуровневые языки программирования 25
Компиляторы 26
Историческая справка 28
1.2. Программирование и решение задач 29
Алгоритмы 29
Разработка программ 30
Объектно-ориентированное программирование 32
Жизненный цикл программного обеспечения 33
1.3. Введение в C++ 34
Как возник язык C++ 34
Пример программы на C++ 35
Ловушка: использование /n вместо \n при переходе на новую строку 38
Совет программисту: синтаксис ввода и вывода 38
Структура простой программы на C++ 39
Ловушка: пробел перед именем файла в директиве include 41
Компиляция и запуск программы на C++ 41
Совет программисту: запуск программы 41
1.4. Тестирование и отладка 43
Виды программных ошибок 44
Ловушка: предполагается, что программа верна 45

Резюме 45
Ответы к упражнениям для самопроверки 46
Практические задания 48

Глава 2. Основные понятия C++ 50

2.1. Переменные и операторы присваивания 50
Переменные 50
Имена и идентификаторы 52
Объявление переменных 54
Операторы присваивания 55
Ловушка: неинициализированные переменные 56
Совет программисту: используйте информативные имена 58
2.2. Ввод-вывод 58
Вывод с помощью потока cout 59
Директивы include и пространства имен 60
Управляющие последовательности 61
Совет программисту: заканчивайте каждую программу выводом символа новой строки 63
Форматирование чисел с дробной частью 63
Ввод с помощью потока cin 64
Программирование ввода и вывода 66
Совет программисту: переводы строки при вводе-выводе 66
2.3. Типы данных и выражения 67
Типы int и double 67
Другие числовые типы 69
Тип char 70
Тип bool 72
Совместимость типов данных 72
Арифметические операторы и выражения 73
Ловушка: целые числа и деление 75
Еще об операторах присваивания 77
2.4. Простейшее управление потоком 77
Простой механизм ветвления 78
Ловушка: сравнение нескольких значений 83
Ловушка: использование = вместо == 83
Составные операторы 84
Простые механизмы циклического выполнения 86
Операторы инкрементирования и декрементирования 90
Пример: баланс кредитной карточки 90
Ловушка: бесконечные циклы 92
2.5. Некоторые особенности оформления программ 93
Отступы 94
Комментарии 94
Именованные константы 96

Резюме 98
Ответы к упражнениям для самопроверки 99
Практические задания 103

Глава 3. Процедурная абстракция и функции 106

3.1. Нисходящее проектирование 106
3.2. Стандартные функции 107
Использование функций 107
Преобразование типов 111
Старая форма оператора приведения типов 113
Ловушка: при целочисленном делении отбрасывается дробная часть 113
3.3. Функции, определяемые программистом 114
Определения функций 115
Две формы объявлений функций 120
Ловушка: неверный порядок аргументов 120
Синтаксис определения функции 122
Еще об определениях функций 123
3.4. Процедурная абстракция 124
Аналогия с черным ящиком 124
Совет программисту: выбор имен формальных параметров 126
Пример: покупка пиццы 126
Совет программисту: использование псевдокода 132
3.5. Локальные переменные 133
Аналогия с маленькой программой 133
Пример: расчет предполагаемого урожая 135
Глобальные константы и глобальные переменные 135
Формальные параметры, передаваемые по значению, являются переменными 137
Пространство имен std 139
Пример: функция, вычисляющая факториал 141
3.6. Перегрузка имен функций 142
Основные понятия перегрузки 143
Пример: модифицированная программа покупки пиццы 145
Автоматическое преобразование типов 147
Резюме 149
Ответы к упражнениям для самопроверки 150
Практические задания 153

Глава 4. Функции для разных подзадач 156

4.1. Функции типа void 156
Определения функций типа void 157
Пример: преобразование значений температуры 158
Оператор return в функциях типа void 160

4.2. Передача параметров по ссылке 162
Первое знакомство с передачей параметров по ссылке 162
Механизм передачи параметров по ссылке 165
Пример: функция swap_values 167
Смешанные списки параметров 168
Совет программисту: как правильно выбрать способ передачи параметра 169
Ловушка: локальная переменная вместо параметра, передаваемого по ссылке 170
4.3. Использование процедурной абстракции 173
Функции, вызывающие другие функции 173
Предусловия и постусловия 173
Пример: цены в супермаркете 176
4.4. Тестирование и отладка функций 180
Заглушки и отладочные программы 180
Резюме 185
Ответы к упражнениям для самопроверки 185
Практические задания 188

Глава 5. Потоки ввода-вывода 192

5.1. Потоки и основы файлового ввода-вывода 192
Почему для ввода-вывода используются файлы 193
Файловый ввод-вывод 194
Введение в классы и объекты 197
Совет программисту: проверяйте, открыт ли файл 199
Реализация файлового ввода-вывода 201
Добавление данных в файл (факультативный материал) 204
Использование имен файлов в качестве входных данных (факультативный материал) 205
5.2. Дополнительные средства выполнения потокового ввода-вывода 208
Форматирование выходных данных с помощью потоковых функций 208
Манипуляторы 211
Потоки в качестве аргументов функций 214
Совет программисту: проверка на конец файла 216
О пространствах имен 217
Пример: форматирование файла 218
5.3. Символьный ввод-вывод 219
Функции-члены get и put 220
Функция-член putback (факультативный материал) 222
Пример: проверка входных данных 224
Ловушка: лишний символ \n во входном потоке 225
Функция-член eof 228
Пример: редактирование текстового файла 230
Предопределенные символьные функции 232
Ловушка: функции toupper и tolower возвращают значения типа int 234

5.4. Наследование 235
Наследование для потоковых классов 236
Пример: еще одна функция new_line 239
Аргументы функции, используемые по умолчанию (факультативный материал) 240
Резюме 242
Ответы к упражнениям для самопроверки 243
Практические задания 249

Глава 6. Определение классов 254

6.1. Структуры 254
Структуры для разнородных данных 254
Ловушка: отсутствие точки с запятой в определении структуры 258
Структуры как аргументы функций 259
Совет программисту: пользуйтесь иерархическими структурами 260
Инициализация структур 261
6.2. Классы 264
Определение классов и функций-членов 264
Открытые и закрытые члены класса 269
Совет программисту: объявляйте все переменные-члены как закрытые 274
Совет программисту: определяйте аксессоры и мутаторы 275
Совет программисту: используйте для объектов оператор присваивания 277
Пример: класс BankAccount 278
Резюме: некоторые свойства классов 282
Инициализация с помощью конструкторов 284
Совет программисту: всегда определяйте конструктор, используемый по умолчанию 289
Ловушка: конструкторы без аргументов 291
6.3. Абстрактные типы данных 293
Классы как абстрактные типы данных 293
Пример: альтернативная реализация класса BankAccount 296
Резюме 301
Ответы к упражнениям для самопроверки 301
Практические задания 307

Глава 7. Поток управления программы 310

7.1. Использование логических выражений 310
Вычисление логических выражений 310
Ловушка: логические выражения преобразуются в значения типа int 314
Функции, возвращающие логические значения 316
Перечисления (факультативный материал) 317
7.2. Многонаправленное ветвление 318
Вложенные операторы if...else 318
Совет программисту: используйте скобки во вложенных операторах 319

Многонаправленные операторы if...else 321
Пример: подоходный налог штата 323
Оператор switch 326
Ловушка: забытый оператор break в операторе switch 329
Использование операторов switch для создания меню 329
Совет программисту: использование в операторах ветвления вызовов функций 331
Блоки 331
Ловушка: переменные, случайно оказавшиеся локальными 334
7.3. Циклы в C++ 335
Цикл while 335
Операторы инкрементирования и декрементирования 337
Оператор for 340
Ловушка: лишняя точка с запятой в операторе for 344
Каким циклом пользоваться 344
Ловушка: неинициализированные переменные и бесконечные циклы 346
Оператор break 347
Ловушка: оператор break во вложенных циклах 348
7.4. Разработка циклов 349
Циклы для сумм и произведений 349
Завершение циклов 350
Вложенные циклы 353
Отладка циклов 357
Резюме 359
Ответы к упражнениям для самопроверки 360
Практические задания 366

Глава 8. Дружественные функции и перегрузка операторов 370

8.1. Дружественные функции 370
Пример: функция равенства 370
Применение дружественных функций 373
Совет программисту: определяйте и аксессоры, и дружественные функции 376
Совет программисту: используйте и функции-члены, и обычные функции 378
Пример: класс Money 379
Реализация функции digit_to_int (факультативный материал) 384
Ловушка: ведущие нули в числовых константах 385
Квалификатор const 387
Ловушка: непоследовательное использование квалификатора const 388
8.2. Перегрузка операторов 391
Реализация перегрузки операторов 392
Конструкторы для автоматического приведения типов 395
Перегрузка унарных операторов 396
Перегрузка операторов >> и << 398

Резюме 406
Ответы к упражнениям для самопроверки 406
Практические задания 413

Глава 9. Раздельная компиляция и пространства имен 416

9.1. Раздельная компиляция 416
Еще раз об абстрактных типах данных 417
Пример: DigitalTime — отдельно компилируемый класс 418
Директива #ifndef 427
Совет программисту: определение других библиотек 429
9.2. Пространства имен 430
Пространства имен и директива using 430
Создание пространств имен 432
Уточнение имен 434
Особенности использования пространств имен (факультативный материал) 435
Безымянные пространства имен 436
Совет программисту: выбор имени для пространства имен 441
Ловушка: не путайте глобальное и безымянное пространства имен 441
Резюме 443
Ответы к упражнениям для самопроверки 443
Практические задания 445

Глава 10. Массивы 447

10.1. Основные понятия 447
Объявление массивов и доступ к их элементам 447
Совет программисту: используйте для работы с массивами цикл for 449
Ловушка: элементы массивов всегда нумеруется начиная с нуля 450
Совет программисту: задавайте размер массивов с помощью определенных в программе констант 450
Расположение массивов в памяти 450
Ловушка: выход индекса массива за допустимые пределы 452
Инициализация массивов 453
10.2. Массивы и функции 455
Элементы массива в качестве аргументов функций 455
Массивы в качестве аргументов функций 457
Квалификатор параметра const 460
Ловушка: несогласованное использование квалификатора const 462
Функции, возвращающие массивы 463
Пример: диаграмма производительности 463
10.3. Использование массивов 475
Частично заполненные массивы 475
Совет программисту: не ограничивайте количество формальных параметров 478
Пример: поиск в массиве 479
Пример: сортировка массива 481

10.4. Массивы и классы 485
Массивы структур и массивы классов 485
Массивы как члены классов 489
Пример: класс для частично заполненного массива 489
10.5. Многомерные массивы 493
Основные понятия 493
Параметры типа многомерных массивов 494
Пример: программа, использующая двухмерный массив 495
Ловушка: запятые между индексами массива 499
Резюме 499
Ответы к упражнениям для самопроверки 500
Практические задания 506

Глава 11. Строки и векторы 515

11.1. Массивы для хранения строк 515
Строковые значения и строковые переменные C 516
Ловушка: использование операторов = и == со строками C 519
Функции из библиотеки cstring 521
Ввод и вывод строк C 524
Преобразование строк C в числа 527
11.2. Стандартный класс string 532
Первое знакомство с классом string 532
Ввод-вывод с помощью класса string 534
Совет программисту: дополнительные версии функции getline 537
Ловушка: комбинирование ввода с помощью потока cin и функции getline 538
Обработка строк с помощью класса string 539
Пример: проверка палиндрома 542
Взаимное преобразование объектов типа string и строк C 546
11.3. Векторы 547
Основные понятия 547
Ловушка: индекс в квадратных скобках превышает размер вектора 550
Совет программисту: учитывайте особенности присваивания для векторов 550
Эффективное использование памяти 550
Резюме 552
Ответы к упражнениям для самопроверки 552
Практические задания 554

Глава 12. Указатели и динамические массивы 558

12.1. Указатели 558
Переменные-указатели 559
Основы управления памятью 565
Ловушка: зависшие указатели 566

Динамические и автоматические переменные 566
Совет программисту: определяйте типы указателей 566
12.2. Динамические массивы 568
Массивы переменных и переменные-указатели 569
Создание и использование динамических массивов 570
Арифметические операции с указателями (факультативный материал) 574
Многомерные динамические массивы (факультативный материал) 576
12.3. Классы и динамические массивы 578
Пример: класс для строковой переменной 578
Деструкторы 582
Ловушка: указатели как параметры, передаваемые по значению 584
Конструктор копирования 585
Перегрузка оператора присваивания 590
Резюме 592
Ответы к упражнениям для самопроверки 593
Практические задания 595

Глава 13. Рекурсия 598

13.1. Рекурсивные функции, не возвращающие значений 599
Пример: вертикальная запись чисел 599
Подробно о рекурсии 604
Ловушка: бесконечная рекурсия 605
Стеки и рекурсия 607
Ловушка: переполнение стека 609
Рекурсия и итеративное выполнение 609
13.2. Рекурсивные функции, возвращающие значения 610
Схема рекурсивных функций, возвращающих значения 610
Пример: еще одна функция возведения в степень 611
13.3. Рекурсивное мышление 615
Рекурсивные технологии проектирования 615
Пример: реализация двоичного поиска с помощью рекурсии 616
Пример: рекурсивная функция-член 623
Резюме 626
Ответы к упражнениям для самопроверки 627
Практические задания 630

Глава 14. Шаблоны 633

14.1. Шаблоны для абстрактных алгоритмов 633
Шаблоны функций 634
Ловушка: сложности компиляции 637
Пример: универсальная функция сортировки 639
Совет программисту: как определять шаблоны 643
Ловушка: использование шаблона с неподходящим типом данных 643
14.2. Шаблоны и абстракция данных 644
Синтаксис шаблона класса 644
Пример: класс-массив 647

Резюме 650
Ответы к упражнениям для самопроверки 651
Практические задания 654

Глава 15. Указатели и связные списки 656

15.1. Узлы и связные списки 656
Узлы 656
Связные списки 661
Вставка узла в начало списка 662
Ловушка: потерянные узлы 664
Поиск в связном списке 665
Указатели в качестве итераторов 668
Вставка и удаление элементов в середине списка 669
Ловушка: использование оператора присваивания с динамическими структурами данных 672
15.2. Применение связных списков 673
Стеки 673
Пример: класс Stack 674
Резюме 678
Ответы к упражнениям для самопроверки 678
Практические задания 680

Глава 16. Наследование и полиморфизм 683

16.1. Наследование 683
Производные классы 684
Конструкторы в производных классах 691
Ловушка: использование закрытых переменных-членов базового класса 693
Ловушка: закрытые функции-члены можно считать ненаследуемыми 694
Модификатор protected 695
Переопределение функций-членов 697
Переопределение и перегрузка 700
Доступ к переопределенной функции базового класса 700
16.2. Подробнее о наследовании 701
Функции, которые не наследуются 702
Операторы присваивания и конструкторы копирования в производных классах 702
Деструкторы и производные классы 703
16.3. Полиморфизм 705
Позднее связывание 705
Виртуальные функции в языке C++ 706
Виртуальные функции и расширение совместимости типов 711
Ловушка: проблема расщепления 714
Ловушка: без функций-членов не обойтись 714
Ловушка: попытка откомпилировать определение класса без определений всех его виртуальных функций-членов 715
Совет программисту: объявляйте деструкторы как виртуальные 716

Резюме 717
Ответы к упражнениям для самопроверки 718
Практические задания 722

Глава 17. Обработка исключений 725

17.1. Основы обработки исключений 726
Простой пример обработки исключений 726
Определение собственных классов исключений 733
Выброс и перехват исключений различных типов 735
Ловушка: сначала перехватывайте специализированные исключения 737
Совет программисту: классы исключений могут быть пустыми 738
Выброс исключений в функции 738
Спецификация исключений 740
Ловушка: спецификация исключений в производных классах 741
17.2. Программирование обработки исключений 742
Когда следует выбрасывать исключения 743
Ловушка: необработанные исключения 744
Ловушка: вложенные блоки try...catch 744
Ловушка: злоупотребление исключениями 744
Иерархия классов исключений 745
Проверка наличия свободной памяти 745
Повторный выброс исключения 746
Резюме 746
Ответы к упражнениям для самопроверки 746
Практические задания 747

Приложение 1. Ключевые слова C++ 749

Приложение 2. Приоритет выполнения операторов 750

Приложение 3. Набор символов ASCII 752

Приложение 4. Некоторые библиотечные функции 753

Приложение 5. Макрос assert 759

Приложение 6. Встраиваемые функции 760

Приложение 7. Перегрузка квадратных скобок индекса массива 761

Приложение 8. Указатель this 763

Приложение 9. Перегрузка операторов как операторов-членов 766

Алфавитный указатель 768

Предисловие

Данная книга представляет собой учебник по программированию на языке C++ для начинающих. При работе с ней читателю не требуется какой-либо опыт программирования либо глубокие математические знания, превышающие уровень средней школы. Однако следует отметить, что и более опытные программисты найдут здесь немало полезного материала.

В США учебник выдержал уже четыре издания, которыми воспользовались сотни тысяч студентов и преподавателей. Предлагаемая читателю книга является переводом последнего четвертого издания. В ней автор, получивший множество отзывов и полезных предложений, после их тщательного анализа сделал ряд изменений и дополнений, и теперь выражает надежду, что в таком виде учебник максимально соответствует предъявляемым к нему требованиям. Нынешнее издание отвечает стандартам ANSI/ISO и полностью адаптировано для использования компиляторов, соответствующих новому стандарту C++.

Последовательность изучения курса

Большинство учебников по C++ для начинающих требуют соблюдения строгой последовательности изложения материала. Эта книга построена иначе: здесь можно менять порядок рассмотрения глав и разделов без потери связности представленных тем. Для работы с учебником вам потребуются только стандартные библиотеки, которые поставляются со всеми реализациями C++.

Структура книги позволяет гибко подойти к освещению классов. В начале процесса обучения вы можете либо ограничиться использованием только стандартных классов, либо сразу рассматривать разработку и применение собственных классов. Имеющийся опыт преподавания показывает, что раннее изучение классов так же полезно и эффективно, как и раннее изучение функций. Конечно, этот материал не назовешь простым, но и к концу курса овладевать им будет ничуть не легче, чем в начале. Более того, изучение классов на первой стадии обучения позволяет лучше освоить эту тему и «привыкнуть» к ним как к базовым программным элементам. Автор рекомендует новичкам работать с учебником в порядке следования глав, те же, кто уже имеет опыт программирования на C++, могут организовать работу по какой-либо иной удобной для себя схеме.

Порядок изложения материала в главах

Для того чтобы написать хороший учебник, мало изложить темы в нужном порядке. Более того, недостаточно даже, чтобы материал был освещен ясно и правильно с точки зрения преподавателя или опытного специалиста. Изложение должно быть понятным и доступным для восприятия в первую очередь начинающим программистам, и именно такую цель ставил перед собой автор. Отзывы многих студентов, пользовавшихся предыдущими изданиями учебника, подтвердили, что эта цель достигнута, и даже более того, книга получилась интересной, а чтение ее доставляет удовольствие тем, кто действительно хочет освоить предмет.

Ниже рассказано, по какому принципу построены главы учебника.

Материал каждой главы содержит необходимую информацию по какой-либо теме, касающейся программирования на языке C++. В тексте приведено большое количество врезок, где кратко повторяются основные термины и определения. Кроме того, автор снабдил каждую главу примечаниями, акцентирующими внимание читателя на каких-либо нюансах программирования (они оформлены в виде разделов «Ловушка» и «Совет программисту»).

Стандартный курс программирования и вычислительной техники часто включает не обязательные для изучения темы, и даже если они не являются частью курса, их полезно вводить в учебники в качестве дополнительных. Данная книга содержит много факультативного материала, который можно интегрировать в курс или оставить для самостоятельного изучения желающими.

Все важнейшие выводы, сделанные в процессе подробного освещения очередной темы, кратко излагаются в конце каждой главы в виде резюме.

Все главы содержат разделы с упражнениями для самопроверки, выполнение которых позволяет закрепить изложенный материал и проверить степень его усвоения. В конце каждой главы приведены ответы к этим упражнениям.

Кроме того, все главы включают практические задания – те проекты, которые вы можете реализовать самостоятельно, опираясь на изученный материал.

Где получить дополнительные материалы

В дополнение к данной книге вы можете использовать материалы, доступные по адресу http://www.aw.com/savitch/, а в качестве альтернативы загрузить программы с сайта автора книги по адресу http://www-cse.ucsd.edu/users/savitch/.

От издательства

Свои замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение о книге!

Подробную информацию о книгах издательств «Питер» и «Издательская группа BHV» вы найдете на веб-сайтах http://www.piter.com и http://www.bhv.kiev.ua.

Глава 1 Основы информатики и программирования на языке C++

Любые аналитические операции теперь можно выполнять при помощи машин ...
Создание Аналитической Машины определит все дальнейшее развитие науки.

Чарльз Бэббидж (1792-1871)

В этой главе описываются основные составляющие компьютера, а также базовые технологии разработки и написания программ. В конце главы мы рассмотрим простую программу на языке C++ и разберемся, как он работает.

1.1. Компьютерные системы

Набор инструкций, которые выполняет компьютер, называется *программой*, а все множество используемых компьютером программ — *программным обеспечением* компьютера. Реальные физические машины и устройства, составляющие компьютер, именуются *аппаратным обеспечением*. Вы увидите, что внутреннее строение компьютеров не такое уж и сложное, поэтому для вас не составит труда изучить его в общих чертах. Но в настоящее время компьютеры снабжают огромным количеством программного обеспечения, предназначенного для создания новых программ. Это разнообразные редакторы, трансляторы и управляющие программы, которые вместе составляют среду разработки программ, являющуюся сложной и мощной системой. Данная книга посвящена программному обеспечению, но краткйй обзор основ устройства и работы аппаратного обеспечения не будет лишним.

Аппаратное обеспечение

Существует три базовых класса компьютеров: персональные компьютеры, рабочие станции и мэйнфреймы. *Персональные компьютеры* (*ПК*) — это относительно

маломощные компьютеры, которые могут использоваться одновременно только одним человеком. *Рабочая станция* на порядок мощнее, хотя и ее можно считать персональным компьютером, поскольку она тоже используется одним человеком. Наконец, *мэйнфрейм* – это еще более мощный компьютер с которым, как правило, одновременно работает несколько человек; для его обслуживания требуется специально обученный персонал. Приведенная терминология довольно часто употребляется, когда речь идет о вычислительной технике, однако при ее применении все же не следует забывать, что разделение на классы условно и четкой границы между ними нет.

Сеть представляет собой группу компьютеров, соединенных друг с другом для совместного использования информации и аппаратных ресурсов (например, принтеров). Сеть может состоять из некоторого количества рабочих станций и одного или более мэйнфреймов, а также совместно используемых внешних устройств.

С точки зрения программирования не имеет значения, работаете вы на ПК, мэйнфрейме или рабочей станции. Далее будет рассказано, что конфигурация компьютерных систем в общих чертах одинакова для всех классов компьютеров.

Аппаратное обеспечение большинства компьютерных систем организовано так, как показано на рис. 1.1. Компьютер может быть условно разделен на пять основных составляющих: устройство (или устройства) ввода, устройство (или устройства) вывода, процессор (называемый также ЦПУ – центральное процессорное устройство), основная память и вторичная память. Процессор, основная память, а зачастую и вторичная память, обычно располагаются в одном корпусе. Процессор и основная память являются базовой частью компьютера, их можно условно считать одним устройством. Остальные компоненты соединены с основной памятью и работают под управлением процессора. Стрелки на рис. 1.1 указывают направление информационных потоков.

Устройством ввода называется любое устройство, позволяющее человеку передавать информацию в компьютер. Наиболее часто используемыми устройствами ввода являются клавиатура и мышь.

Устройством вывода именуется любое устройство, с помощью которого компьютер передает информацию человеку. Обычно это дисплей, называемый также *монитором*. Очень часто к компьютеру подключено более одного устройства вывода. Так, в дополнение к монитору компьютер обычно имеет принтер для печати выходной информации на бумаге. Слово *терминал* иногда употребляют для обозначения совокупности устройств ввода и вывода компьютерной системы, например клавиатуры и монитора.

Для хранения входной информации и результатов вычислений в состав компьютера включается *память*. Выполняемая компьютером программа также хранится в памяти. Память компьютера делится на два вида: основную (называемую также оперативной) и вторичную. Выполняемая программа находится в оперативной памяти, которая играет наиболее важную роль в работе компьютера. *Оперативная память* состоит из длинного списка нумерованных ячеек, называемых *ячейками памяти*; их количество в разных компьютерах различно и варьируется от нескольких тысяч до многих миллионов, а иногда даже и миллиардов. Каждая ячейка

памяти содержит строку нулей и единиц — то есть некоторое *двоичное число*. Одна цифра такого числа называется *двоичной цифрой* или *битом* и представляет собой единицу либо нуль. Содержимое ячеек может изменяться компьютером, поэтому ячейку памяти можно представить в виде крошечной школьной доски, на которой можно бесконечно писать и стирать написанное. В большинстве компьютеров все ячейки памяти содержат одинаковое количество битов, как правило, восемь (или некоторое их количество, кратное восьми). Ячейка памяти размером 8 бит именуется *байтом* и имеет свой уникальный номер. Оперативную память компьютера с этой точки зрения можно рассматривать как длинный список пронумерованных байтов. Число, идентифицирующее конкретный байт, называют его *адресом*. Элемент данных, например число или буква, может храниться в одном из байтов памяти, и адрес этого байта используется для его поиска.

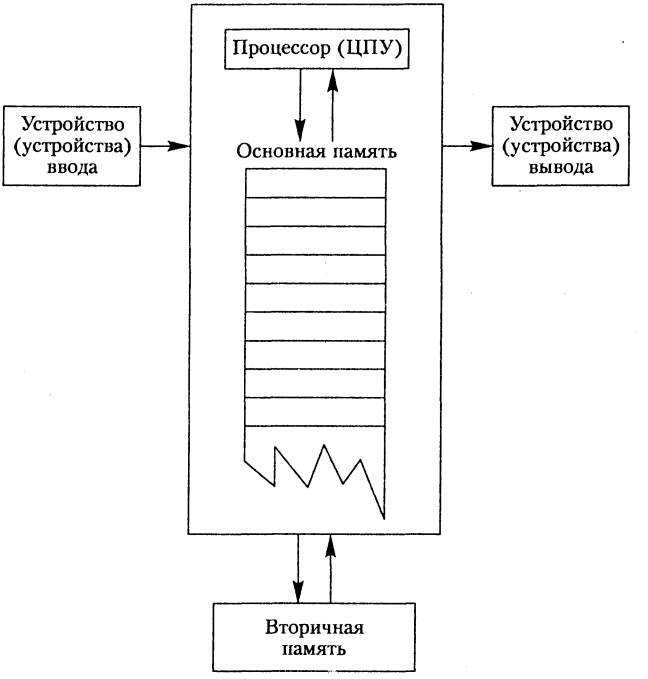


Рис. 1.1. Главные компоненты компьютера

Если компьютер должен работать с элементом данных, требующим для хранения более одного байта (скажем, с большим числом), этот элемент записывается в несколько последовательно расположенных в памяти байтов. Начальным адресом области памяти, выделенной для хранения элемента данных, считается адрес ее первого байта. Таким образом, оперативную память компьютера можно представлять как набор областей памяти разного размера, при этом размер каждой области выражается в байтах, а ее начальным адресом является адрес ее первого байта. На рис. 1.2 показана гипотетическая структура оперативной памяти компьютера. Размер этих областей памяти не фиксирован и при выполнении компьютером очередной программы может изменяться.

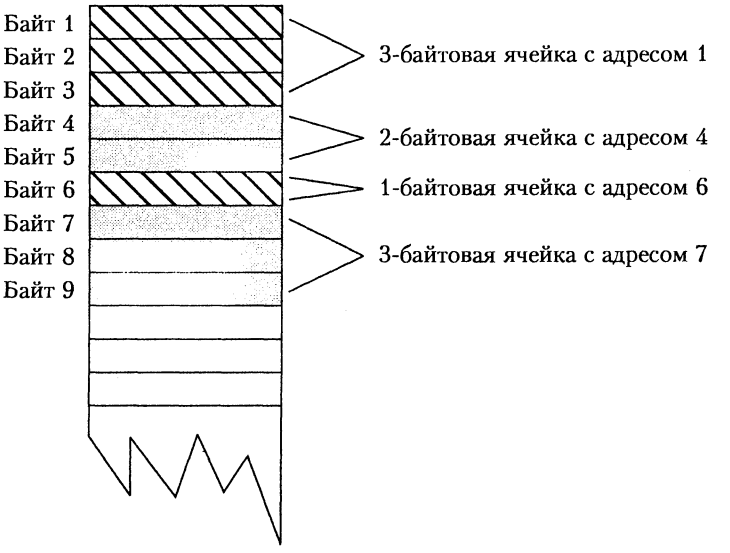


Рис. 1.2. Области памяти и байты

Байты и адреса

Оперативная память делится на множество нумерованных ячеек, именуемых *байтами*. Номер байта называется его *адресом*. Для хранения отдельно взятого элемента данных, такого как число или буква, служит группа последовательно расположенных в памяти байтов. Начальным адресом области памяти, выделенной для хранения элемента данных, является адрес ее первого байта.

Тот факт, что информация в памяти компьютера представлена в виде нулей и единиц, не имеет большого значения для программирования на языке C++ (как и на многих других языках). Однако знать это все же необходимо по следующей причине. Компьютер интерпретирует последовательности нулей и единиц как числа, буквы, инструкции или данные других типов. Такую интерпретацию он выполняет автоматически в соответствии с определенными схемами кодирования. При этом для каждого типа данных используется своя схема: буквы кодируются одним способом, целые числа — другим, дробные — третьим, команды — четвертым и т. д. В частности, в одной из наиболее распространенных кодировок последовательность 01000001 представляет букву A, а также число 65. Чтобы узнать, что именно представляет эта последовательность в памяти, хранящей конкретный элемент данных, компьютер должен знать, какая схема кодирования использовалась при записи такой последовательности. К счастью, программисту редко приходится иметь дело с такими кодами, и в большинстве случаев он может считать, что в памяти хранятся буквы, числа или другие нужные ему данные.

До сих пор мы с вами говорили только об оперативной памяти, без которой компьютер не может выполнить ни одной операции. Однако этот вид памяти используется только тогда, когда компьютер выполняет инструкции программы. У него

имеется и другая память, именуемая вторичной памятью или *вторичным запоминающим устройством*. (Слова «память» и «запоминающее устройство» в данном контексте являются синонимами.) *Вторичная память* — это память, которая используется для постоянного хранения данных по окончании (или до начала) работы компьютера, ее также называют *внешней памятью* или *внешним запоминающим устройством*.

Во внешнем запоминающем устройстве информация хранится в виде блоков, называемых *файлами*, которые могут быть очень большими или маленькими — такими, как создаст их пользователь. Например, в файле на внешнем запоминающем устройстве может храниться программа, а когда ее нужно выполнить, она копируется в оперативную память. Помимо программ в файлах могут содержаться произвольные данные: письма, инвентаризационные ведомости — короче, все что угодно.

С одним компьютером может быть соединено несколько разных типов внешних запоминающих устройств. Наиболее распространенными являются жесткие диски, дискеты и компакт-диски. (Дискеты иногда называют *гибкими дисками*.) Используемые в компьютерах *компакт-диски* (CD) почти ничем не отличаются от музыкальных компакт-дисков. Они могут быть доступными только для чтения (чтобы компьютер мог считывать, но не изменять записанную на них информацию) или для чтения и записи (тогда компьютер может модифицировать хранящиеся на них данные). Хранение информации на жестких дисках и дискетах принципиально ничем не отличается от хранения на компакт-дисках. *Жесткие диски* закреплены в компьютере, и обычно их нельзя извлечь. *Дискеты* и компакт-диски легко достать из дисковода и перенести на другой компьютер. Их преимущество в том, что они недороги и портативны, тогда как достоинство жестких дисков в возможности хранения большего количества данных и более быстрой работе. Помимо этих трех наиболее распространенных носителей, с которыми вы будете постоянно иметь дело, существуют и другие виды внешней памяти, о которых в настоящей книге не рассказывается.

Почему восемь

Байт — это ячейка памяти размером восемь бит. Почему именно восемь? Тому есть две причины. Во-первых, число 8 является степенью числа 2 (8 — это 2^3). Поскольку компьютеры оперируют битами, имеющими всего два возможных значения, степени числа 2 для них удобнее степеней числа 10. Во-вторых, для кодирования одного символа (буквы или другого символа, вводимого посредством клавиатуры) необходимо восемь бит, то есть один байт.

Оперативную память часто называют *ОЗУ* (*оперативное запоминающее устройство*) или *RAM* (от англ. random access memory — *память с произвольным доступом*). Под *произвольным доступом* понимается непосредственный доступ к данным по задаваемому адресу. Внешняя память обычно требует *последовательного доступа*, при котором в поиске нужных данных компьютер просматривает значительную часть памяти или всю память от ее начала до месторасположения этих данных.

Процессор, называемый также *центральным процессорным устройством* (*ЦПУ*), – это «мозг» компьютера. Рекламируя новый компьютер, его производитель, прежде всего, сообщает о том, на базе какого процессора он создан. Процессор представляет собой электронную *микросхему* (или *чип*). Он следует инструкциям программы и выполняет заданные в них вычисления. Но процессор является очень простым «мозгом», все, что он может, – выполнять заданный программистом набор несложных инструкций. Типичные инструкции процессора таковы: «интерпретировать нули и единицы как числа и сложить число, хранящееся в памяти по адресу 37, с числом по адресу 59, а ответ поместить по адресу 42» или же «прочитать букву на входе, преобразовать ее в код, состоящий из нулей и единиц, и поместить этот код в память по адресу 1298». Процессор может складывать, вычитать, умножать, делить, перемещать данные из одних ячеек памяти в другие, а также интерпретировать строки нулей и единиц как буквы и отправлять их на устройство вывода. Кроме того, он обладает примитивной способностью реорганизовывать инструкции программы. Набор поддерживаемых процессором инструкций зависит от его типа. Процессоры современных компьютеров поддерживают до нескольких сотен инструкций, но все они так же просты, как только что описанные.

Программное обеспечение

Обычно человек взаимодействует с компьютером не непосредственно, а через операционную систему. *Операционная система* выделяет ресурсы для разных выполняемых компьютером задач. Она представляет собой программу, которую можно представить себе как начальника группы служащих. Операционная система руководит всеми служебными программами и передает им ваши запросы. Если нужно запустить какую-нибудь программу, вы передаете операционной системе имя файла, в котором она содержится, и операционная система запускает эту программу. Если требуется отредактировать файл, вы сообщаете операционной системе его имя, и она запускает редактор. Для большинства пользователей операционная система и является компьютером, поскольку компьютера без операционной системы они никогда не видели. Наиболее распространенными операционными системами являются UNIX, DOS, Linux, Windows, Macintosh и VMS.

Программа – это набор инструкций, предназначенных для выполнения компьютером. Как показано на рис. 1.3, входную информацию компьютера можно разделить на две части: программу и данные. Компьютер следует инструкциям программы и выполняет некоторые операции. *Данные* – это то, что можно концептуально определить как входную информацию. Например, если программа складывает два числа, то эти числа являются данными. Можно рассматривать вопрос и так: данные являются входной информацией для программы, а программа и данные – это входная информация для компьютера (обычно вводимая посредством операционной системы). Предоставляя компьютеру программу и предназначенные для нее данные, мы говорим, что *запускаем* программу с указанными данными, а компьютер *выполняет* ее с ними. Слово «данные» имеет и более широкий смысл, чем в приведенном выше определении, – оно обозначает любую доступную компьютеру информацию. Используется оно одинаково часто как в узком, так и в широком смысле.

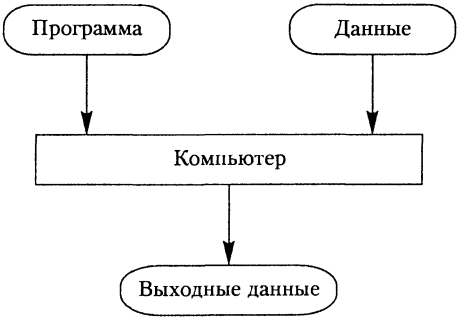


Рис. 1.3. Упрощенная схема выполнения программы

Высокоуровневые языки программирования

Существует множество языков программирования. Эта книга посвящена написанию программ на языке C++, который является высокоуровневым языком программирования, как и многие другие (C, Java, Pascal, Visual Basic, FORTRAN, COBOL, Lisp, Scheme и Ada). *Высокоуровневые языки программирования* во многом напоминают человеческие языки. Они разработаны так, чтобы человеку как можно легче было создавать на них программы и читать их. Инструкции высокоуровневого языка программирования гораздо сложнее тех простых инструкций, которые может выполнять центральный процессор компьютера.

Языки, близкие по структуре к языку инструкций процессора, называются языками низкого уровня. Они ориентированы на конкретные компьютеры, поэтому наборы их инструкций для разных компьютеров различны. Типичная инструкция языка низкого уровня такова:

```
ADD X Y Z
```

Она может означать следующее: «прибавить число, хранимое в области памяти, обозначенной X, к числу, хранимому в области памяти, обозначенной Y, и поместить результат в область памяти, обозначенную Z». Эта простая инструкция написана на так называемом *языке ассемблера*. Хотя язык ассемблера очень близок к языку, который понимает компьютер, созданные на нем программы перед выполнением требуют некоторого простого преобразования. Чтобы компьютер мог выполнить ассемблерную инструкцию, ее нужно *транслировать* в последовательность нулей и единиц. В частности, слово `ADD` может быть преобразовано в 0110, `X` в 1001, `Y` в 1010 и `Z` в 1001. После такой трансляции получится следующая компьютерная инструкция:

```
0110 1001 1010 1001
```

Инструкции языка ассемблера и их эквиваленты, состоящие из нулей и единиц, для разных компьютеров различны.

О понятных компьютеру программах в форме последовательностей нулей и единиц говорят, что они написаны на *машинном языке* (*машинном коде*). Язык ассемблера и машинный язык почти одинаковы (не считая, конечно, представления

инструкций), и разница между ними для нас не важна. Нам принципиально важно различие между машинным языком и языками высокого уровня, подобными C++. Заключается оно в том, что программа на языке высокого уровня должна быть преобразована (транслирована) в машинный код, и только тогда компьютер сможет ее понять и выполнить.

Компиляторы

Программа, выполняющая трансляцию языка высокого уровня, такого как C++, в машинный код, называется *компилятором*. То есть компилятор представляет собой особый вид программы, для которой входными данными является другая программа, а выходными — преобразованная входная программа. Во избежание путаницы входная программа обычно называется *исходной программой* или *исходным кодом*, а ее транслированная версия, сгенерированная компилятором, именуется *объектной программой* или *объектным кодом*. Слово *код* часто используется для обозначения программы или части программы, и особенно часто оно употребляется в отношении объектных программ. Теперь предположим, что вы хотите запустить программу, написанную на языке C++. Чтобы компьютер мог понять ее инструкции, нужно выполнить следующее. Прежде всего запустите компилятор и задайте в качестве его входных данных программу на C++. В этом случае компилятор воспримет ее не как набор инструкций, а просто как длинную строку символов. Его выходными данными будет другая длинная строка символов, представляющая собой эквивалент исходной программы, написанный на машинном языке. Далее запустите программу на машинном языке, передав ей входные данные, предназначавшиеся для программы на языке C++. Выходные данные, которые вы в результате получите, можно рассматривать как выходные данные программы на C++. Описанный процесс представлен в виде схемы на рис. 1.4. При ее изучении имейте в виду, что показанные на схеме два компьютера на самом деле представляют один компьютер, используемый дважды: первый раз для трансляции программы на языке C++, а второй раз для выполнения результирующей программы на машинном языке.

Компилятор

Компилятор — это программа, транслирующая программу, написанную на языке высокого уровня, например на C++, в понятную компьютеру программу на машинном языке, которую он может непосредственно выполнить.

Реальный процесс трансляции и выполнения программы на языке C++ несколько сложнее, чем показано на рис. 1.4. В любой программе на C++ используются операции (такие, как процедуры ввода и вывода), которые уже запрограммированы, и их не нужно в каждой программе кодировать вновь. Более того, они уже откомпилированы, и их объектный код может быть соединен с объектным кодом вашей программы для получения полной программы на машинном языке, предназначенной для выполнения компьютером. Это соединение выполняет специальная программа, называемая *компоновщиком*. Ее задача — объединить заданный

набор фрагментов объектного кода в одну исполняемую программу. Как взаимодействуют компилятор и компоновщик, показано на рис. 1.5. Компоновку процедур и других простых фрагментов кода многие системы выполняют автоматически.

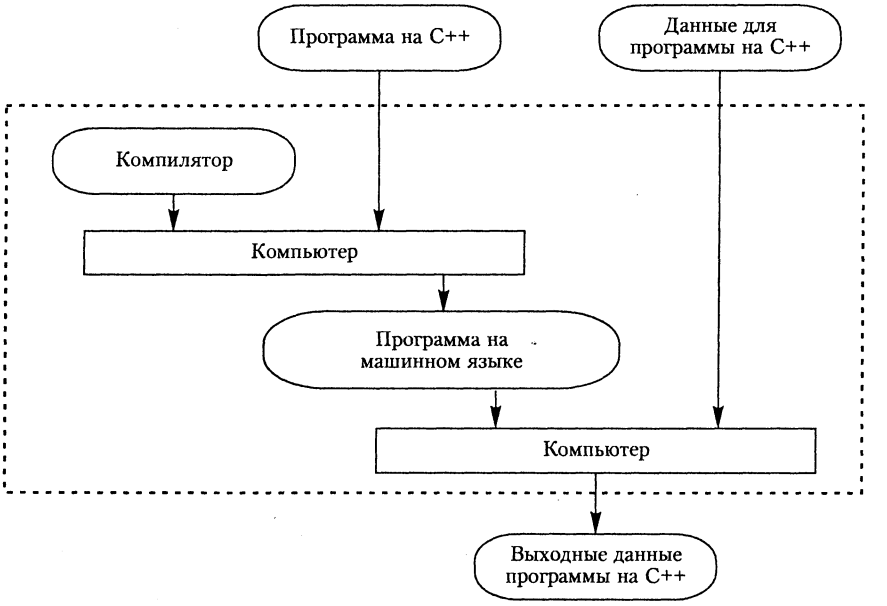


Рис. 1.4. Компиляция и запуск программы на C++ (упрощенная схема)

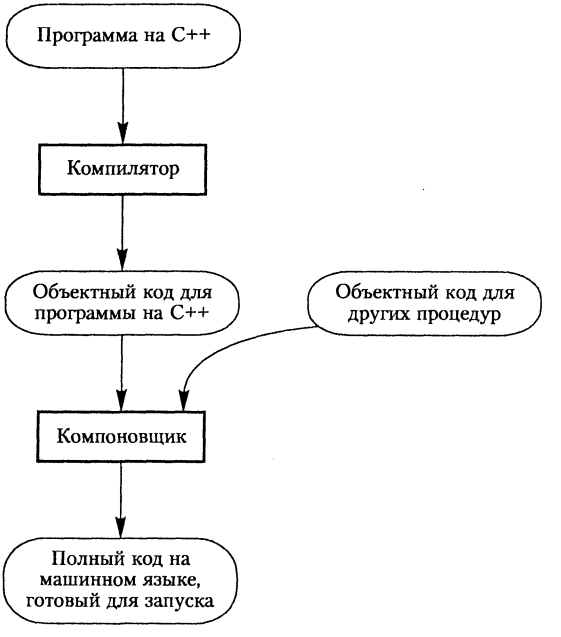


Рис. 1.5. Подготовка программы на C++ для запуска

Компоновка

Объектный код программы на C++ должен быть объединен с объектным кодом используемых ею процедур (таких, как процедуры ввода и вывода). Это процесс называется *компоновкой* и осуществляется программой, именуемой *компоновщиком*. Для простых программ компоновка может выполняться автоматически.

Упражнения для самопроверки

1. Назовите пять основных составляющих компьютера.
2. Какими должны быть входные данные программы, складывающей два числа?
3. Какими должны быть входные данные программы, выставляющей студентам оценки за тесты?
4. В чем состоит различие между программами на машинном языке и на языке высокого уровня?
5. Каково назначение компилятора?
6. Что такое исходная программа? Что такое объектная программа?
7. Что такое операционная система?
8. Каково назначение операционной системы?
9. Как называется установленная на компьютере операционная система, с помощью которой вы будете готовить свои программы?
10. Что такое компоновка?
11. Выясните, как выполняет компоновку используемый вами компилятор: автоматически или нет.

Историческая справка

Первым программируемым компьютером можно считать «аналитическую машину», которую разработал английский физик и математик Чарльз Бэббидж. Известно, что ученый начал работу несколько ранее 1822 года и продолжал ее до конца жизни. Хотя создание машины так никогда и не было завершено, именно с нее началась история компьютерной науки. Многое из того, что мы знаем о Чарльзе Беббидже и конструкции «аналитической машины», нам известно из работ его коллеги, Ады Августы, которую часто называют первым компьютерным программистом. Ада Августа была дочерью поэта Байрона, графиней Лавлейс. Ее высказывание о процессе решения задач с помощью компьютера, процитированное в начале следующего раздела, актуально и сегодня. В компьютерах нет ничего волшебного, и они, по крайней мере до сих пор, не способны находить решения всех задач, с которыми сталкивается человек. Они просто делают то, что велит программист. Решение задачи выполняется компьютером, но постановка задачи и описание способа ее решения возлагаются на человека. Именно с данного вопроса мы и начнем наш рассказ о компьютерном программировании.

1.2. Программирование и решение задач

Аналитическая Машина не претендует на роль источника чего бы то ни было. Она может делать что угодно, но только если мы можем определить для нее эти действия. Она может выполнить анализ; но не в состоянии самостоятельно осознавать какие бы то ни было отношения или истины. Ее роль — помогать нам получить то, что нам уже известно.

Ада Августа, графиня Лавлейс (1815-1852)

В этом разделе рассматриваются общие принципы разработки и написания программ. Данные принципы относятся не только к C++, но и к любому другому языку программирования.

Алгоритмы

При изучении первого языка программирования вам может показаться, что самой трудной частью решения задачи с помощью компьютера является перевод идей на доступный ему язык. На самом же деле это совершенно не так. Самым трудным является поиск способа решения. Когда он найден, его перевод на язык программирования, будь то C++ или любой другой язык, — рутинное дело, требующее определенных навыков, но не более того. Поэтому полезно временно забыть о языке программирования и полностью сконцентрироваться на том, чтобы сформулировать шаги решения задачи и записать их на естественном языке в виде инструкций, адресованных человеку, а не компьютеру.

Последовательность точных инструкций для решения задачи называется *алгоритмом*. Близки к этому термину слова «рецепт», «метод», «указания», «процедура». Инструкции алгоритма могут быть выражены на языке программирования или на естественном языке. Мы будем писать алгоритмы и на русском языке, и на языке C++. Поскольку компьютерная программа — это просто алгоритм, выраженный на понятном компьютеру языке, термин «алгоритм» имеет более общий смысл, чем термин «программа». Но говоря, что последовательность инструкций составляет алгоритм, мы обычно имеем в виду, что эти инструкции приведены на естественном языке, так как если бы они были записаны на языке программирования, мы называли бы их программой. Чтобы понять это, рассмотрим пример.

Ниже приведен алгоритм, выраженный в виде инструкций на русском языке. С алгоритмами именно такого типа мы будем иметь дело в данной книге. При помощи этого короткого и простого алгоритма можно определить, сколько раз заданное название встречается в списке. Список содержит названия команд — победителей соревнований по футболу последнего сезона, и в нем нужно найти вашу любимую команду и определить количество ее выигрышей.

Инструкции, пронумерованные от 1 до 5, должны выполняться в порядке их следования. Если не оговорено другое, мы всегда будем предполагать, что инструкции алгоритма выполняются в той последовательности, в которой они записаны.

Но более сложные алгоритмы могут некоторым образом изменять порядок выполнения инструкций, например задавать многократное повторение определенной группы инструкций, как указано в пункте 4 приведенного алгоритма.

Алгоритм, определяющий, сколько раз заданное название встречается в списке

1. Получить список названий.
2. Получить искомое название.
3. Обнулить счетчик.
4. Для каждого элемента списка сравнить название из списка с искомым названием и в случае их совпадения увеличить значение счетчика на единицу.
5. Сообщить пользователю итоговое значение счетчика, которое и будет ответом.

Слово «алгоритм» имеет длинную историю. Оно происходит от имени математика и астронома Аль-Хорезми, жившего в VIII-IX веках в Персии. Аль-Хорезми написал известный трактат о числах и равенствах, озаглавленный «Китаб аль-джебр валь-мукабала», что переводится как «Книга о восстановлении и противопоставлении». От арабского слова «аль-джебр» (восстановление) и произошло современное название «алгебра». Когда-то термины «алгебра» и «алгоритм» были связаны друг с другом более тесно, нежели сейчас. В те времена слово «алгоритм» употреблялось только в отношении алгебраических правил решения числовых уравнений. Сегодня же под ним могут подразумеваться самые разнообразные последовательности правил, которые описывают операции не только с числовыми, но и с любыми другими (например, символьными) данными. Последовательность инструкций может быть названа алгоритмом только в том случае, если она полностью и однозначно определяет некоторые действия и порядок их выполнения.

Алгоритм

Алгоритм — это последовательность точных инструкций, выполнение которых приводит к решению поставленной задачи.

Разработка программ

Разработка программы часто является достаточно сложной задачей. Для ее выполнения не существует четких правил, и ее нельзя описать каким-либо алгоритмом, потому что это творческий процесс. И все же при создании программы обычно придерживаются схематического плана, показанного на рис. 1.6. Как видите, весь процесс разработки делится на две фазы: фазу решения задачи и фазу реализации этого решения. Результатом *фазы решения задачи* является алгоритм решения задачи, записанный на естественном языке. Для создания программы на языке программирования, таком как C++, алгоритм переводится на этот язык. Процесс создания конечной программы не основе алгоритма называется *фазой реализации*.

Первым делом вам необходимо убедиться, что задача, которую должна выполнять ваша программа, определена однозначно и в полном объеме. К этому шагу следует отнестись максимально серьезно – если вы толком не представляете себе, что хотите получить на выходе программы, то результаты могут вас удивить. Вам следует знать совершенно точно, какие данные должны быть заданы на входе программы, какие получены на выходе и в какой форме они будут представлены. Например, если речь идет о программе для работы с банковскими счетами, нужно знать не только процентную ставку, но и то, как будут начисляться проценты: ежегодно, ежемесячно, ежедневно или с иной периодичностью. Если программа должна писать стихи, вам необходимо знать требуемый стихотворный размер: будет это белый стих, ямб, гекзаметр или какая-то иная форма.

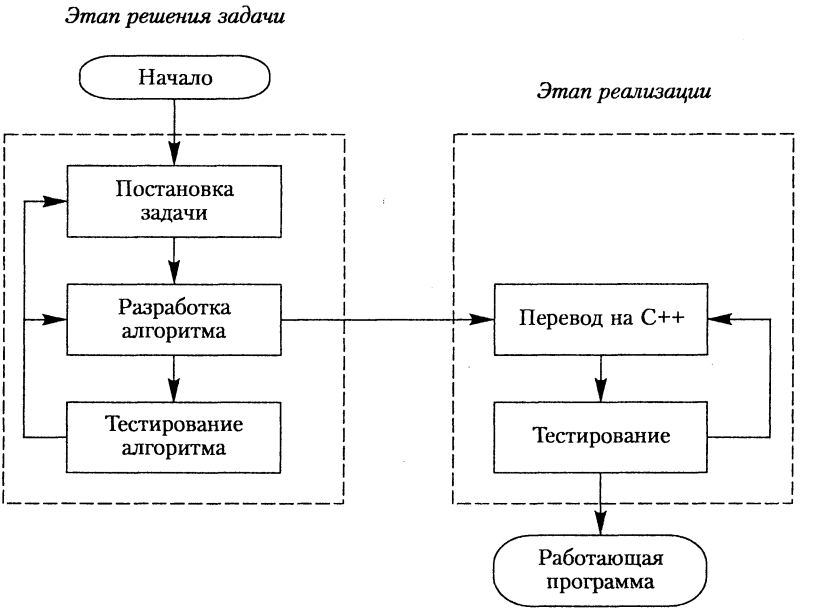


Рис. 1.6. Процесс разработки программы

Многие программисты-новички не понимают, для чего нужно разрабатывать алгоритмы до написания программы на языке программирования, и пытаются ускорить процесс, полностью опустив этап решения задачи или сократив его до определения задачи. На самом деле, почему бы не направиться прямо к цели и не сберечь время? Дело в том, что такая экономия себя не оправдывает! Практика показывает, что, напротив, процесс, выполняемый в два этапа, позволяет быстрее получить правильно работающую программу. Таким образом упрощается этап разработки алгоритма, снимаются ограничения, которые накладывает синтаксис конкретного языка программирования, и в результате вероятность ошибок сводится к минимуму. Но даже для программы относительно небольшого размера день тщательной проработки алгоритма поможет сэкономить несколько дней утомительного поиска ошибок в неверно составленной программе.

Как показано на рис. 1.6, тестирование выполняется на обоих этапах. Перед написанием программы проверяется алгоритм, и если он оказывается неадекватным, программист его дорабатывает или модифицирует. В простейшем случае проверка производится просто путем устного анализа и выполнения всех указанных в нем действий, а для алгоритмов большего размера применяются карандаш и бумага. Программа на языке C++ тестируется путем ее компиляции и выполнения при использовании тестовых входных данных. Одни ошибки компилятор обнаруживает самостоятельно и выдает соответствующие сообщения, а другие находит программист, осуществляя проверку правильности выходных данных.

На рис. 1.6 показана упрощенная схема разработки программы. Реальный процесс более сложен, ошибки и несоответствия обнаруживаются неожиданно, а иногда приходится возвращаться к уже выполненным шагам и переделывать работу. Например, при тестировании алгоритма может оказаться, что постановка задачи неполная, тогда придется вернуться и доработать ее. Дефекты постановки задачи или алгоритма могут обнаружиться и позднее, во время тестирования программы. В этом случае нужно вернуться к постановке задачи или алгоритму, после чего внести коррективы в соответствующие элементы программы.

Объектно-ориентированное программирование

Для разработки программы с использованием описанного выше способа мы представляем ее в виде алгоритма (последовательности инструкций), который определяет выполнение операций с некоторыми данными. Этот подход правилен, но не всегда оптимален. При создании современных программ часто применяется концепция, называемая *объектно-ориентированным программированием* или *ООП*. В объектно-ориентированном программировании программа рассматривается как набор взаимодействующих объектов. Эту методику проще всего понять на примере программы, моделирующей какой-нибудь процесс. Так, в программе, моделирующей движение автомобилей по шоссе, объекты могут представлять автомобили и полосы шоссе. Для каждого объекта создают алгоритмы, описывающие его поведение в разных ситуациях. Суть объектно-ориентированного программирования заключается в разработке объектов и используемых ими алгоритмов. В этом контексте этап разработки алгоритма, показанный на рис. 1.6, должен быть заменен этапом разработки объектов и их алгоритмов.

Основными характеристиками ООП являются инкапсуляция, наследование и полиморфизм. *Инкапсуляцию* обычно определяют как форму сокрытия информации или абстрагирования. Это правильное, но малопонятное определение, поэтому проще сказать, что инкапсуляция – это способ упрощения определения объекта. *Наследование* обеспечивает возможность написания многократно используемого программного кода. *Полиморфизм* позволяет связать с одним именем несколько значений в контексте наследования. Употребляя термины объектно-ориентированного программирования, мы хорошо понимаем, что для человека, ничего до сих пор о нем не слышавшего, такие термины совершенно непонятны. Но единственной нашей целью было их упомянуть, а определения и подробные описания базовых концепций ООП будут приведены позднее. Язык C++ поддерживает

объектно-ориентированное программирование, позволяя тем самым разработчику определять и использовать классы – особый тип данных, сочетающий в себе данные и алгоритмы.

Жизненный цикл программного обеспечения

Разработчики больших программных систем, таких как компиляторы и операционные системы, делят процесс разработки программного обеспечения на шесть перечисленных ниже этапов, которые составляют *жизненный цикл программного обеспечения*.

1. Анализ и постановка задачи.
2. Проектирование программного обеспечения (создание объектов и разработка алгоритмов).
3. Реализация (написание программного кода).
4. Тестирование.
5. Сопровождение и дальнейшая доработка системы.
6. Моральное старение системы.

Когда мы говорили о разработке программного обеспечения, то последние два этапа не упоминали, так как они наступают по окончании работы над программой и после ввода ее в эксплуатацию. Однако забывать о них не следует. Программа должна проектироваться так, чтобы ее легко было читать и изменять, поскольку в дальнейшем она, наверняка, потребует доработки. Разработка программы с учетом ее дальнейшей модификации является очень важной темой, которую мы подробно рассмотрим позднее, когда у нас появится для этого достаточная база. Что касается морального старения, то тут все просто: когда требования к программе изменяются и ее не удается модифицировать с разумными затратами, она выводится из эксплуатации и заменяется совершенно новой программой.

Упражнения для самопроверки

12. Алгоритм подобен кулинарному рецепту, однако он не может включать некоторые действия, перечисляемые в рецепте. Какие из указанных ниже действий допустимы в алгоритме:

    а) положить две чайные ложки сахару в емкость миксера;

    б) добавить одно яйцо;

    в) добавить одну чашку молока;

    г) добавить унцию рому, если вы не за рулем;

    д) добавить ваниль по вкусу;

    е) взбивать до однородности;

    ж) вылить в красивый стакан;

    з) посыпать мускатным орехом.

13. Назовите действие, которое нужно выполнить первым в процессе создания программы.
14. Процесс разработки программы можно разделить на два основных этапа. Что это за этапы?
15. Объясните, почему нельзя пренебрегать этапом решения задачи.

1.3. Введение в C++

Язык — только инструмент науки ...

Сэмюель Джонсон (1709-1784)

В этом разделе вы получите начальное представление о языке программирования C++, которому посвящена наша книга.

Как возник язык C++

Первое, чем обратил на себя внимание язык C++, это его необычное название. Что это за язык такой? Может быть C? Существуют ли языки C- или C--? Есть ли языки с названиями A и B? Ответы на большинство этих вопросов отрицательны. На большинство, но не на все. Существует язык B, предшественником которого является не язык A, а язык, называемый BCPL. Язык C был разработан на основе языка B, а C++, в свою очередь, на основе языка C. Почему в названии языка C++ используются два символа «плюс»? Как вы узнаете из следующей главы, «++» — это одна из операций, поддерживаемых языками C и C++, так что получается хороший каламбур. Языки BCPL и B нас не интересуют — это просто ранние версии языка программирования C. А рассказ о языке C++ мы начнем с языка C.

Язык программирования C был разработан в 1970-х годах Деннисом Ричи в AT&T Bell Laboratories. На нем была написана операционная система UNIX. (Эта операционная система существовала и до появления C, но ее первые версии создавались либо на языке ассемблера, либо на B — языке, разработанном Кеном Томпсоном, создателем UNIX.) C — язык общего назначения, подходящий для написания любых программ, но его успех и популярность тесно связаны именно с операционной системой UNIX. Если вам нужно модифицировать что-то в UNIX, для этого потребуется C. Они изначально были связаны друг с другом так тесно, что со временем не только системные программы, но и все коммерческое программное обеспечение, работающее под управлением UNIX, было написано на языке C. Этот язык стал настолько популярным, что были созданы его версии и для других распространенных операционных систем. Теперь его использование уже не ограничивается компьютерами, на которых установлена ОС UNIX.

Несмотря на огромную популярность C обладает рядом недостатков. Это довольно необычный язык в том отношении, что, являясь языком высокого уровня, он содержит много элементов языка низкого уровня. Поэтому C, скорее, находится посередине между двумя крайностями — языком очень высокого и очень низкого уровня, причем в этом состоит его сила и его слабость. С одной стороны, подобно низкоуровневому языку ассемблера, C позволяет напрямую манипулировать памятью

компьютера, а с другой стороны, благодаря использованию элементов высокоуровневого языка, в нем очень упрощены такие операции, как чтение и запись данных. Так что язык C прекрасно подходит для написания системных приложений, а для программ прикладного характера он менее удобен, поскольку сложнее других языков высокого уровня, и программы на нем менее читабельны. Кроме того, он выполняет значительно меньше автоматических проверок и других полезных операций.

Для преодоления такого рода ограничений в начале 1980-х годов Бьярном Страуструпом из AT&T Bell Laboratories был разработан язык C++. В него вошла большая часть элементов языка C, поэтому большинство программ, написанных на C, являются программами на C++. (Обратное неверно: программы на C++ в подавляющем большинстве не являются программами на C.) В отличие от C, язык C++ включает средства для объектно-ориентированного программирования – очень мощной технологии, разработанной относительно недавно.

Пример программы на C++

В листинге 1.1 приведена простая программа на C++ и показано, что будет выведено на экран дисплея после ее выполнения. Вводимый *пользователем* текст выделен в примере полужирным шрифтом, чтобы он отличался от текста, выводимого программой (*пользователем* называется человек, запускающий программу). Но на самом деле и тот, и другой текст отображается на экране одинаковым шрифтом. Человек, написавший программу, называется *программистом*. Роли этих двух людей ни в коем случае не следует путать – пользователь и программист могут быть двумя разными людьми или одним и тем же лицом. Скажем, если вы написали программу и запустили ее, то являетесь и программистом и пользователем. Но, как правило, программа создается одними людьми, а используется другими.

В следующей главе мы подробно расскажем о тех элементах языка C++, которые применяются для написания программ, аналогичных приведенной в листинге 1.1. Пока же рассмотрим эту программу, чтобы в общих чертах разобраться, что такое язык C++. И пусть вас не смущает, если некоторые детали будут вам не ясны.

Листинг 1.1. Пример программы на C++

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

int main()
{
    int number_of_pods, peas_per_pod, total_peas;

    cout << "Press Enter after entering a number.\n";
    cout << "Enter the number of pods:\n";
    cin >> number_of_pods;
    cout << "Enter the number of peas in a pod:\n";
    cin >> peas_per_pod;

    total_peas = number_of_pods * peas_per_pod;

    cout << "If you have ";
    cout << number_of_pods;
```

продолжение

Листинг 1.1 (*продолжение*)

```
cout << " pea pods\n";
    cout << "and ";
    cout << peas_per_pod;
    cout << " peas in each pod, then\n";
    cout << "you have ";
    cout << total_peas;
    cout << " peas in all the pods.\n";

    return 0;
}
```

Пример диалога

```
Press Enter after entering a number.
Enter the number of pods:
10
Enter the number of peas in a pod:
9
If you have 10 pea pods
and 9 peas in each pod, then
you have 90 peas in all the pods.
```

Начало и конец приведенной программы содержат детали, которые мы в этой главе рассматривать не будем. Программа начинается такими строками:

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

int main()
{
```

Пока будем считать, что они просто означают: «здесь программа начинается».

Завершают программу следующие строки:

```
    return 0;
}
```

Они означают: «здесь программа заканчивается».

Строки, расположенные между этими двумя фрагментами кода, являются главной частью программы. Давайте рассмотрим их, начиная со строки

```
int number_of_pods, peas_per_pod, total_peas;
```

Эта строка называется *объявлением переменных*. Она сообщает компьютеру, что number_of_pods, peas_per_pod и total_peas будут использоваться в качестве имен трех *переменных*. О том, что такое переменные, мы подробно расскажем в следующей главе, но их роль в нашей программе достаточно очевидна: в рассматриваемом случае переменные используются для обозначения чисел. Слово int (сокращение от англ. integer — целочисленный), с которого начинается строка, сообщает компьютеру, что числа, идентифицируемые этими переменными, являются целыми (такими, как 1, 2, –1, –7, 0, 205, –103 и т. п.).

Остальные строки представляют собой инструкции, указывающие компьютеру выполнять какие-нибудь действия. Они именуются *операторами* или *исполняемыми операторами*. В приведенной программе каждый оператор занимает в точности одну строку. Однако так бывает не всегда — операторы могут быть длинными и занимать несколько строк.

Большинство операторов программы начинается со слова cin или cout. Эти операторы отвечают за ввод и вывод соответственно. Слово cin (читается «си-ин») используется для обозначения ввода. Операторы, начинающиеся с этого слова, говорят компьютеру, что делать, когда информация вводится с клавиатуры. Слово cout (читается «си-аут») используется для обозначения вывода, то есть отправки информации из программы на экран терминала. Буква c, с которой начинаются эти слова, обозначает язык C++. Последовательности символов, представляющие собой удвоенные знаки «больше» и «меньше», указывают направление перемещения данных. Символ << обозначает запись или помещение значения в указанный слева приемник, а символ >> — выборку или извлечение значения из указанного слева источника. В качестве примера рассмотрим такую строку:

```
cout << "Press Enter after entering a number.\n";
```

Ее можно понять так: «поместить "Press Enter after entering a number.\n" в cout» или просто «вывести "Press Enter after entering a number.\n"». Если считать, что слово cout является именем экрана (выходного устройства), то стрелки указывают компьютеру вывести заданную в кавычках строку на экран. Символы \n в конце строки непосредственно на экран не выводятся — они только говорят компьютеру, что после вывода текста следует перейти на новую строку. Следующий оператор программы тоже начинается со слова cout; он выводит на экран такую строку:

```
Enter the number of pods:
```

Потом в программе идет строка, начинающаяся со слова cin, — это оператор ввода:

```
cin >> number_of_pods;
```

Данную строку можно понимать так: «прочитать number_of_pods из cin» или просто «ввести number_of_pods».

Если считать, что слово cin является обозначением клавиатуры (устройства ввода), то знак >> указывает на то, что ввод должен быть направлен с клавиатуры в переменную number_of_pods. Посмотрите еще раз на пример диалога, приведенный после листинга 1.1. В третьей строке полужирным шрифтом выделено число 10. Это выделение (по принятому в нашей книге соглашению) показывает, что число введено с клавиатуры. После нажатия клавиши Enter (иногда называемой Return) оно станет доступно программе. Оператор, начинающийся с cin, указывает компьютеру переслать входное значение 10 в переменную number_of_pods. С этого момента переменная number_of_pods будет иметь значение 10; и когда далее в программе мы будем встречать эту переменную, она будет обозначать для нас число 10.

Рассмотрим следующие две строки:

```
cout << "Enter the number of peas in a pod:\n";
cin >> peas_per_pod;
```

Они похожи на две предыдущие. Первая выводит на экран сообщение с просьбой ввести число. Когда вы вводите число с клавиатуры и нажимаете клавишу Enter, это число становится значением переменной peas_per_pod. В приведенном примере диалога пользователь ввел число 9. После его ввода и нажатия клавиши Enter значение переменной peas_per_pod становится равным 9.

Затем идет строка программы, которая выполняет вычисления:

```
total_peas = number_of_pods * peas_per_pod;
```

Символ звездочки (*) в языке C++ обозначает умножение. Данный оператор говорит, что нужно умножить number_of_pods на peas_per_pod. То есть 10 умножается на 9 и в результате получается 90. Знак равенства имеет особое значение (не такое, как в математике): он указывает, что переменной, расположенной слева от него, должно быть присвоено значение. В нашем случае переменной total_peas присваивается значение 90.

Остальная часть программы должна быть вам уже в некоторой степени понятна — она выводит данные на экран. Следующие три строки:

```
cout << "If you have ";
cout << number_of_pods;
cout << " pea pods\n";
```

являются просто операторами вывода, работающими точно так же, как предыдущие операторы, начинавшиеся со слова cout. Нечто новое содержит только вторая из приведенных строк, выводящая переменную number_of_pods. При выводе переменной на экран выводится ее значение. Поэтому в данном случае на экране появляется число 10. (Напомним, что в нашем примере запустивший программу пользователь присвоил переменой number_of_pods значение 10.) Таким образом, приведенные три строки выводят на экран следующее:

```
If you have 10 pea pods
```

Обратите внимание на то, что все три фрагмента помещены в одной строке. Новая строка не будет начата до тех пор, пока в выводимой строке не встретится символ \n.

В остальной части программы не содержится ничего нового, и если вы разобрались в материале этого раздела, то без труда ее поймете.

Ловушка: использование /n вместо \n при переходе на новую строку

При использовании в операторе, начинающемся со слова cout, сочетания \n будьте внимательны: в нем применяется обратная косая. Если по ошибке написать не \n, а /n, компилятор не выдаст сообщения об ошибке и ваша программа будет работать, но выведет не те результаты, которых вы ожидали.

Совет программисту: синтаксис ввода и вывода

Если считать cin именем клавиатуры, или устройства ввода, а cout именем экрана, или устройства вывода, очень легко запомнить, какие знаки использовать — «больше» или «меньше», поскольку они указывают направление перемещения данных. В качестве примера рассмотрим такой оператор:

```
cin >> number_of_pods;
```

Здесь данные перемещаются с клавиатуры в переменную number_of_pods и знак >> указывает направление от cin к переменной.

А в операторе вывода

```
cout << number_of_pods;
```

данные перемещаются из переменной number_of_pods на экран и знак << указывает направление от переменной к cout.

Структура простой программы на C++

Структура простой программы на языке C++ представлена в листинге 1.2. С точки зрения компилятора разрывы строк и отступы не обязательно должны быть такими, как в наших примерах, поскольку ему все равно, где и как они расположены. А вот с точки зрения пользователя программа должна быть написана именно так, чтобы ее удобно было читать. Когда открывающая и закрывающая фигурные скобки расположены на отдельных строках, их легко найти. Отступы перед операторами и их размещение на отдельных строках также улучшают читабельность программы. Далее в этой книге будут встречаться операторы, не помещающиеся на одной строке — для них будет использоваться другое оформление с применением отступов и разрывов строк.

В листинге 1.1 объявления переменных располагались в строке, начинавшейся со слова int. Однако, как вы увидите в следующей главе, вовсе не обязательно помещать все объявления переменных в начало программы. Это, скорее, общепринятое соглашение, а поскольку оно очень удобно, лучше всего ему следовать (как в листингах 1.1 и 1.2), если только у вас нет особой причины поступать иначе. Операторы — это инструкции, выполняемые компьютером. В листинге 1.1 операторами являются строки с первыми словами cout и cin, а также одна строка, начинающаяся с идентификатора total_peas, за которым следует знак равенства. Операторы также называют исполняемыми операторами — мы будем использовать оба термина как равнозначные. Обратите внимание, что каждый оператор оканчивается точкой с запятой. Этот символ имеет примерно тот же смысл, что и точка в предложении: он отмечает конец оператора.

Листинг 1.2. Структура простой программы на C++

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

int main()
{
    объявления_переменных

    оператор_1
    оператор_2
       ...
    последний_оператор

    return 0;
}
```

Первые несколько строк мы пока будем рассматривать как способ указать на начало программы. Но кое-что в них можно объяснить подробнее. Строка

```
#include <iostream>
```

называется *директивой* include. Она сообщает компилятору, где он может найти информацию о некоторых используемых в программе элементах. В данном случае iostream является именем библиотеки, которая содержит определения процедур, выполняющих ввод с клавиатуры и вывод на экран; это также имя файла, хранящего необходимую информацию об этой библиотеке. Программа-компоновщик объединяет объектный код библиотеки iostream с объектным кодом написанной вами программы. Для библиотеки iostream в вашей системе это, скорее всего, будет сделано автоматически. В дальнейшем вы будете пользоваться и другими библиотеками, и тогда их имена тоже нужно будет указать компилятору с помощью директив include в начале программы. Для иных библиотек одной только названной директивы может оказаться недостаточно, но сама она является обязательной. Директивы всегда начинаются с символа #. Некоторые компиляторы требуют, чтобы вокруг него не было пробелов, так что лучше всегда помещать этот символ в начало строки и не отделять его пробелом от слова include.

Следующая строка «уточняет» приведенную выше директиву include:

```
using namespace std;
```

Она сообщает, что имена, определенные в iostream, нужно интерпретировать «стандартным образом» (std является сокращением от англ. standard – стандартный). Пока это все, что можно сказать о данной строке.

Третья и четвертая непустые строки просто указывают, что начинается главная часть программы:

```
int main()
{
```

Правильнее было бы сказать «главная функция», а не «главная часть», но смысл термина «функция» будет объяснен только в главе 3. Фигурные скобки { и } отмечают начало и конец главной части программы. Они не обязательно должны располагаться на отдельных строках, но так их легче найти, поэтому мы всегда будем размещать их подобным образом.

Строка

```
return 0;
```

говорит, что здесь программу нужно завершить. Данная строка не обязательно должна быть последней, но в простой программе нет причин размещать ее где-либо в другом месте. Некоторые компиляторы позволяют вообще ее опустить, и считают концом программы последний оператор. Но есть компиляторы, которые требуют, чтобы программа обязательно содержала строку return 0, и поэтому лучше всего взять за правило всегда включать ее в программу. Эта строка называется *оператором возврата* и считается исполняемым оператором, поскольку указывает на действие, которое должен выполнить компьютер. Число 0 пока не имеет для нас никакого значения, но оно обязательно должно присутствовать. Его смысл станет вам понятен по мере изучения материала. Пока же обратите внимание на то,

что хотя оператор return и сообщает о конце программы, главная ее часть должна быть завершена закрывающей фигурной скобкой }.

Ловушка: пробел перед именем файла в директиве include

Следите за тем, чтобы между символом < и именем файла (в рассматриваемом нами случае iostream), а также между именем файла и символом > не было пробела (см. листинг 1.2). Иначе при обработке директивы include компилятор будет искать файл, имя которого начинается или оканчивается пробелом! А не найдя такого файла, он выдаст сообщение об ошибке, которую будет трудно обнаружить. Можете намеренно сделать эту ошибку в маленькой программе, откомпилировать ее, а затем сохранить выданное компилятором сообщение, чтобы в следующий раз знать, что оно означает.

Компиляция и запуск программы на C++

В предыдущем разделе рассказывалось о том, что произойдет, если запустить программу на C++, приведенную в листинге 1.1. Однако где же находится эта программа и как ее запустить?

Программа на языке C++ пишется с помощью текстового редактора точно так же, как любой текстовый документ, будь то отчет о работе, любовное письмо или список покупок. И так же, как любой другой документ, программа сохраняется в файле. Существует множество различных текстовых редакторов, и здесь рассказывать о них мы не будем, а вот о том, как работает ваш редактор, можно узнать из его документации.

Способ компиляции и запуска программы зависит от конкретной используемой системы, так что вам нужно выяснить, как в ней вызываются команды компиляции, компоновки и запуска программы на C++. Список команд можно найти в документации системы, а еще лучше проконсультироваться у специалиста. В ответ на команду откомпилировать программу генерируется перевод программы на машинный язык, называемый объектным кодом программы, который нужно связать (скомпоновать) с готовым объектным кодом стандартных процедур (таких, как процедуры ввода и вывода). Эта компоновка, скорее всего, будет выполнена автоматически, так что беспокоиться о ней не следует. Но в некоторых системах может потребоваться отдельный вызов компоновщика. В этом случае вам тоже следует обратиться к документации или проконсультироваться у более опытного программиста. Когда программа будет готова, останется ее запустить. Команда ее запуска, как и все остальные, зависит от конкретной используемой вами системы.

Совет программисту: запуск программы

Требования разных компиляторов и разных сред разработки к подготовке исходного файла программы на C++ могут несколько отличаться. Воспользуйтесь копией программы, приведенной в листинге. 1.3 (ее можно загрузить из Интернета

или набрать вручную). Откомпилируйте эту программу. Если получите сообщение об ошибке, проверьте введенный вами текст, исправьте ошибки и перекомпилируйте файл. Когда программа будет откомпилирована без ошибок, попробуйте ее запустить.

Листинг 1.3. Тестовая программа на C++

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

int main()
{
    cout << "Testing 1, 2, 3\n";
    return 0;
}
```

Пример диалога

```
Testing 1, 2, 3
```

Если программа откомпилирована и выполнена нормально, то все в порядке. Для работы с остальными примерами книги больше ничего не нужно. Если же что-то прошло не так, прочитайте этот раздел до конца.

Когда программа, вроде бы, выполняется, но вы не видите выведенного ею текста

```
Testing 1, 2, 3
```

вероятнее всего, она его вывела, но он исчез с экрана слишком быстро. Попробуйте добавить в конец программы перед оператором `return 0` следующие строки:

```
char letter;
cout << "Enter a letter to end the program:\n";
cin >> letter;
```

Они остановят программу, чтобы вы могли прочитать выведенный текст. Теперь часть программы в фигурных скобках должна выглядеть так:

```
cout << "Testing 1, 2, 3\n";
char letter;
cout << "Enter a letter to end the program:\n";
cin >> letter;
return 0;
```

Смысл добавленных строк станет вам понятен, когда вы изучите материал главы 2.

В случае если программа вообще не будет компилироваться или запускаться, попробуйте включить в директиву

```
#include <iostream>
```

расширение имени файла `.h`, вот так:

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

Если же это не поможет, удалите строку

```
using namespace std;
```

Когда программа требует задания `iostream.h` вместо `iostream`, это означает, что вы используете старый компилятор C++, и вам следует установить компилятор более поздней версии.

Если после перечисленных действий программа все равно не откомпилируется, проверьте в документации своей версии C++, не требуются ли для консольного ввода-вывода какие-нибудь дополнительные директивы.

И наконец, если все это не даст желаемого результата, проконсультируйтесь у специалистов по C++. Без сомнения, необходимые изменения будут очень просты — нужно только узнать, что именно следует сделать.

Упражнения для самопроверки

16. Если включить в программу на языке C++ оператор

    ```
    cout << "C++ is easy to understand.";
    ```

    он вызовет вывод на экран некоторого текста. Что отобразится на экране в результате выполнения этого оператора?

17. Каково назначение символов `\n` в следующем операторе (см. листинг 1.1):

    ```
    cout << "Enter the number of peas in a pod:\n";
    ```

18. Что делает следующий оператор (см. листинг 1.1):

    ```
    cin >> peas_per_pod;
    ```

19. Что делает следующий оператор (см. листинг 1.1):

    ```
    total_peas = number_of_pods * peas_per_pod;
    ```

20. Каково назначение директивы

    ```
    #include <iostream>
    ```

21. Что неправильно (если что-то неправильно) в каждой из следующих директив `include`:

    а) `#include <iostream >`

    б) `#include < iostream>`

    в) `#include <iostream>`

1.4. Тестирование и отладка

— Триста шестьдесят пять минус один — сколько это будет?
— Триста шестьдесят четыре, конечно.
Шалтай-Болтай поглядел на Алису с недоверием.
— Ну-ка, посчитай на бумажке, — сказал он.

Льюис Кэррол

Процесс поиска ошибок в программе называется *отладкой*. В этом слове нет ничего примечательного, а вот его английский эквивалент, debugging, который буквально переводится как «устранение жуков», и английское название ошибки в программе, bug (читается «баг»), означающее «жук, насекомое», довольно забавны, как и история их происхождения. Контр-адмирал военно-морского флота США Грейс Мюррей Хоппер (1906-1992) была «третьим программистом первого в мире

крупномасштабного цифрового компьютера». Когда Хоппер работала на компьютере Harvard Mark I, созданном под руководством профессора Гарвардского университета Говарда Айкена, в реле попал мотылек и вызвал сбой в работе компьютера. Программисты поместили останки мотылька в лабораторный журнал и записали: «Сегодня обнаружен первый настоящий баг». Этот журнал выставлен в Музее военно-морского флота в Далгрене, штат Виржиния. Такой была первая задокументированная компьютерная ошибка. Когда профессор Айкен, зайдя в комнату, спросил, как продвигаются вычисления, программисты ответили, что они очищают компьютер от насекомых — по-английски «debugging».

Ошибку в программе программисты до сих пор часто называют «багом». В данном разделе рассказывается о трех основных типах программных ошибок и приводится несколько советов, как их исправлять.

Виды программных ошибок

Компилятор обнаруживает определенные виды ошибок и выводит соответствующие сообщения. Он выявляет все *синтаксические ошибки*, называемые так потому, что они представляют собой нарушения синтаксиса (то есть грамматических правил) языка программирования, например пропущенную точку с запятой.

Если компилятор обнаруживает, что программа содержит синтаксическую ошибку, то сообщает, что это за ошибка и где она находится. Получив такое сообщение, можете не сомневаться, что ошибка действительно есть, но в остальном полностью доверять компилятору нельзя. Он делает все, что в его силах, но нередко ошибается, поскольку лишь пытается предположить, что же вы хотели написать на самом деле. Ведь компилятор — всего лишь программа, и он не может читать ваши мысли. Если же ошибок несколько, то неправильное распознавание первой из них обычно приводит к неверному распознаванию всех остальных.

Итак, если компилятор обнаруживает явное нарушение синтаксических правил языка программирования, он выводит *сообщение об ошибке*. Но иногда он выводит *предупреждающе сообщение*, говорящее о том, что вы сделали что-то, не являющееся явным нарушением, но все же несколько необычное и могущее служить признаком вероятной ошибки. Компилятор как бы спрашивает вас: «Вы уверены, что имели в виду именно это?» И на первых этапах изучения языка обращайте внимание на каждое предупреждение и относитесь к нему как к ошибке, пока не сможете сами точно определить, можно ли его проигнорировать.

Существуют и такие виды ошибок, которые компьютерная система может обнаружить только при выполнении программы. Они называются *ошибками периода выполнения*. Большинство компьютерных систем выявляет определенные ошибки периода выполнения и выводит соответствующие сообщения. Многие из этих ошибок происходят при числовых вычислениях. Так, если компьютер пытается разделить число на нуль, это действие обычно интерпретируется как ошибка периода выполнения.

Если компилятор одобрил программу и вы сразу смогли ее запустить, это еще не означает, что она будет работать так, как вам нужно. Помните, что компилятор проверяет только, является ли предоставленная ему программа синтаксически

правильной, но понятия не имеет, делает ли программа то, что вам требуется. Ошибки исходного алгоритма или его преобразования на язык C++ называются *логическими ошибками*. Например, если в программе из листинга 1.1 вы случайно вместо знака + поставите знак *, это будет логической ошибкой. Программа будет нормально откомпилирована и запущена, но выведет неверный ответ. Если вы не получили сообщений об ошибках во время выполнения программы, но при этом она работает неправильно, значит, в ней содержится логическая ошибка. Такие ошибки труднее всего выявлять, поскольку компьютер не выдает никаких сообщений, могущих вам в этом помочь. Да он и не может этого сделать, поскольку программа работает и не выполняет никаких незаконных с его точки зрения действий. А откуда ему знать, совпадают ли они с вашими намерениями?

Ловушка: предполагается, что программа верна

Для того чтобы протестировать новую программу на наличие логических ошибок, нужно запустить ее несколько раз при разных тестовых наборах входных данных и проверить, как она их обрабатывает. Если программа проходит проверку успешно, ей уже можно в некоторой степени доверять, но все же не до конца, так как определенная вероятность ошибки остается: она может проявиться при каких-нибудь других входных данных. Поэтому очень важно, чтобы программа была написана с максимально возможной аккуратностью.

Упражнения для самопроверки

22. Назовите три основных вида программных ошибок.
23. Какие ошибки обнаруживаются компилятором?
24. Пропуск знака препинания (например, точки с запятой) — это ошибка. К какому типу она относится?
25. Если не поставить последнюю закрывающую скобку программы (}), это будет ошибкой. К какому типу она относится?
26. Предположим, программа содержит код, при проверке которого компилятор выдает предупреждение. Что следует делать в этом случае? Что по этому поводу сказано в книге? Что думаете вы сами?
27. Допустим, вы разработали программу, которая должна ежедневно начислять проценты по банковскому вкладу, но по ошибке написали ее так, что она начисляет проценты не ежедневно, а ежегодно. К какому типу относится эта ошибка?

Резюме

- Используемый компьютером набор программ называется программным обеспечением этого компьютера. Совокупность реальных физических устройств, составляющих компьютерную систему, именуется аппаратным обеспечением.

- Пять основных составляющих компьютера таковы: устройство (устройства) ввода, устройство (устройства) вывода, процессор (ЦПУ), а также основная память и вторичная память.
- Компьютер имеет два вида памяти: основную и вторичную. Основная память используется только во время выполнения программы. Вторичная память применяется для хранения данных, которые находятся в компьютере постоянно.
- Основная память делится на множество нумерованных ячеек, называемых байтами. Номер байта именуется его адресом. Часто байты объединяются в группы по несколько байт, образуя небольшие области памяти. В этом случае начальным адресом такой области является адрес ее первого байта.
- Байт состоит из восьми двоичных цифр, каждая из которых может быть либо нулем, либо единицей. Такая цифра называется битом.
- Компилятор — это программа, транслирующая программный код, написанный на языке высокого уровня, в программу на понятном компьютеру языке, которая может выполняться им непосредственно.
- Алгоритм — это последовательность точных инструкций для решения задачи. Алгоритмы можно писать на естественном языке (скажем, русском) или на языке программирования (например, на C++). Но чаще словом «алгоритм» называют последовательность инструкций на естественном языке.
- Прежде чем приступать к написанию программы на C++, следует разработать ее алгоритм (метод решения выполняемой программой задачи).
- Ошибки программирования делятся на три категории: синтаксические, ошибки периода выполнения и логические. Ошибки первых двух типов выявляет компилятор, а ошибки третьего типа приходится находить самостоятельно.
- Отдельные инструкции программы на C++ называются операторами.
- Переменная в программе на C++ может использоваться для обозначения некоторого числа. (Подробнее о переменных рассказывается в следующей главе.)
- Оператор программы на C++, начинающийся с `cout <<`, является оператором вывода, а оператор, начинающийся с `cin >>`, — оператором ввода.

Ответы к упражнениям для самопроверки

1. Пять основных составляющих компьютера таковы: устройства ввода, устройства вывода, процессор (ЦПУ), основная память и вторичная память.
2. Два складываемых числа.
3. Оценки каждого студента по каждому тесту.
4. Программа на машинном языке должна быть создана в такой форме, чтобы компьютер мог ее сразу выполнить. Программа на языке высокого уровня пишется так, чтобы ее легко мог использовать человек. Перед выполнением компьютером программа на языке высокого уровня подлежит преобразованию (трансляции) в программу на машинном языке.

5. Компилятор транслирует программу на языке высокого уровня в программу на машинном языке.
6. Программа на языке высокого уровня, которая служит входными данными для компилятора, называется исходной программой. Транслированная программа на машинном языке, получаемая после компиляции, называется объектной программой.
7. Операционная система — это программа или комплекс совместно работающих программ, которые обеспечивают взаимодействие пользователя и компьютерной системы, а также запуск и выполнение других программ.
8. Назначение операционной системы заключается в выделении ресурсов компьютера разным выполняемым им задачам.
9. Это может быть Macintosh, Windows 2000, Windows XP, VMS, Solaris, SunOS, UNIX (или операционная система UNIX-типа, например Linux). Существует и множество других операционных систем.
10. Объектный код программы на C++ должен быть объединен с объектным кодом используемых в программе стандартных операций (таких, как ввод и вывод). Этот процесс называется компоновкой. Для простых программ компоновка может быть выполнена автоматически.
11. Ответ на этот вопрос зависит от используемого вами компилятора. Большая часть UNIX-компиляторов выполняет компоновку автоматически, как и компиляторы в большинстве интегрированных сред разработки для операционных систем Windows и Macintosh.
12. Инструкции д), е), ж), з) слишком нечеткие, чтобы их можно было использовать в алгоритме. Таким указаниям как «по вкусу», «до однородности» или «красивый» недостает определенности, как и указанию «посыпать», в котором не сказано, сколько именно мускатного ореха нужно высыпать в стакан. Остальные инструкции вполне могут быть использованы в алгоритме.
13. Первое, что нужно сделать для создания программы, это убедиться, что задача, которую она должна выполнять, определена полностью и четко.
14. Этап решения задачи и этап реализации.
15. Опыт показывает, что процесс, состоящий из двух этапов, позволяет быстрее получить правильно работающую программу.
16. `C++ is easy to understand`
17. При выводе данных комбинация символов `\n` указывает компьютеру начать новую строку, чтобы вывести следующий элемент с новой строки.
18. Этот оператор указывает компьютеру прочитать следующее число, введенное с клавиатуры, и поместить его в переменную с именем `peas_per_pod`.
19. Оператор указывает компьютеру перемножить два числа из переменных `number_of_pods` и `peas_per_pod` и поместить результат в переменную `total_peas`.
20. Директива `#include <iostream>` указывает компилятору прочитать файл `iostream`. В этом файле содержатся объявления `cin`, `cout` операторов записи (<<) и чтения (>>), используемых для ввода и вывода. Директива обеспечивает

правильную компоновку объектного кода библиотеки `iostream` с используемыми в программе операторами ввода-вывода.

21. а) лишний пробел после имени файла `iostream` вызывает сообщение об ошибке «файл не найден»;

    б) лишний пробел перед именем файла `iostream` вызывает сообщение об ошибке «файл не найден»;

    в) это правильная директива.

22. Три основных вида программных ошибок таковы: синтаксические ошибки, ошибки периода выполнения и логические ошибки.

23. Компилятор выявляет синтаксические ошибки. Существуют и другие ошибки, не являющиеся синтаксическими. Вы узнаете о них позднее.

24. Синтаксическая ошибка.

25. Синтаксическая ошибка.

26. В книге сказано, что к предупреждениям следует относиться как к ошибкам до тех пор, пока вы сами не сможете решить, можно игнорировать конкретное предупреждение или нет.

27. Логическая ошибка.

Практические задания

1. Наберите программу на языке C++, приведенную в листинге 1.1, используя текстовый редактор. Проследите за тем, чтобы первая строка была введена в точности так, как в листинге. Она должна начинаться с левого края строки без пробелов до и после символа #. Откомпилируйте эту программу и запустите ее. Если компилятор выдаст сообщение об ошибке, исправьте программу и откомпилируйте вновь. Повторяйте это до тех пор, пока компиляция не пройдет успешно без вывода сообщений об ошибках. После этого запустите программу на выполнение.

2. Модифицируйте программу на C++, введенную в задании 1. Измените ее таким образом, чтобы сначала она писала на экране слово `Hello`, а затем продолжала вывод с той же строки и делала все то же самое, что и программа, приведенная в листинге 1.1. Для этого нужно добавить в программу только одну строку. Перекомпилируйте и запустите новую версию программы. Добавьте в программу еще одну строку, для того чтобы в конце она выводила на экран слово `Good-bye`. Не забудьте добавить в последний оператор вывода комбинацию символов `\n`, вот так:

```
cout << "Good-bye\n";
```

   (Некоторые системы требуют обязательного наличия завершающих символов `\n`, поэтому лучше добавить их на всякий случай.) Откомпилируйте и запустите новую версию программы.

3. Модифицируйте программу на C++ из задания 1 или из задания 2. Замените в вашей программе знак умножения * знаком сложения +. Перекомпилируйте и запустите программу. Заметьте, что программа компилируется и запускается абсолютно нормально, но выводит неверный результат с точки зрения задачи, выполняемой предыдущими программами. Это обусловлено тем, что внесенное вами изменение является логической ошибкой.

4. Напишите программу на C++, считывающую два целых числа и выводящую их сумму и произведение. Можете взять за основу программу, приведенную в листинге 1.1, и модифицировать ее. Проследите за тем, чтобы первая строка вашей программы была в точности такой же, как в программе из листинга 1.1. В частности, она должна начинаться с левого края строки без пробелов до и после символа #. Кроме того, не забудьте добавить в последний оператор вывода вашей программы символы \n. Этот последний оператор может быть, например, таким:

   ```
   cout << "This is the end of the program.\n";
   ```

   (Некоторые системы требуют обязательного наличия завершающих символов \n, так что лучше добавить их на всякий случай.)

5. Задача следующего упражнения заключается в том, чтобы составить небольшой каталог типичных синтаксических ошибок и сообщений об ошибках, с которыми сталкивается начинающий программист. Выполняя это упражнение, вы должны уяснить, что означает каждое из наиболее часто встречающихся сообщений.

   В качестве материала для исследования можете воспользоваться любой программой из заданий 1–4.

   Порядок работы должен быть таким: вы намеренно делаете ошибку в программе, компилируете ее, записываете ошибку и сообщение о ней, исправляете ошибку, снова компилируете программу (чтобы быть уверенным, что она исправлена) и вносите следующую ошибку. Таким образом, составляется каталог ошибок и соответствующих сообщений, который вы будете пополнять по мере изучения курса.

   Ниже перечислены ошибки, которые предлагается по очереди внести в программу:

   а) поместите пробел между символом < и словом `iostream`;

   б) удалите один из символов < или > в директиве `include`;

   в) удалите слово `int` из строки `int main();`

   г) удалите или неправильно напишите слово `main`;

   д) удалите одну или обе скобки в строке `int main();`

   е) продолжайте действовать аналогичным образом, намеренно неправильно запишите идентификаторы `cout`, `cin`, удалите один или оба символа < в операторе `cout`, удалите завершающую фигурную скобку и т. д.

Глава 2 Основные понятия C++

Если вы думаете, что компьютерный терминал состоит из громоздкого телевизора и стоящей перед ним пишущей машинки, то ошибаетесь. На самом деле это интерфейс, посредством которого можно связываться со всем миром.

Дуглас Адамс

В этой главе рассказывается об элементах языка C++, необходимых для написания простейших программ, и рассматривается несколько небольших примеров.

2.1. Переменные и операторы присваивания

Понять, как при программировании используются переменные, — значит, понять суть процесса программирования.

Эдсгер Вайб Дейкстра

Программы манипулируют данными, такими как числа и символы. Для их идентификации и хранения в C++ и многих других языках программирования используются программные структуры, называемые *переменными*. Переменные — это основа языка программирования, поэтому, приступая к описанию C++, мы начнем именно с них. В качестве примера рассмотрим программу, приведенную в листинге 2.1, и проанализируем все ее элементы. Общий смысл этой программы должен быть вам понятен, но некоторые ее детали новы и требуют пояснения. Программа запрашивает у пользователя количество конфет в пакете и вес одной конфеты, а затем определяет и выводит вес всего пакета.

Переменные

В C++ переменные могут хранить числовые данные или данные других типов. Пока мы будем говорить только о первых из них. Такие переменные чем-то напоминают школьную доску, где можно написать любое число. И подобно тому, как

число на доске можно стереть и написать вместо него другое, в C++ в любой момент можно изменить содержимое переменной. Но в отличие от доски, которая вообще может не содержать записей, переменная всегда что-нибудь хранит (это может быть даже «мусор», который остался в памяти компьютера от предыдущей программы). Число или данные другого типа, хранящиеся в переменной, называются ее *значением*. В рассматриваемой нами программе из листинга 2.1 используются три переменные: number_of_bars, one_weight и total_weight. Когда программа выполняется при входных данных, указанных в примере диалога (см. листинг 2.1), значение переменной number_of_bars с помощью оператора

```
cin >> number_of_bars;
```

устанавливается равным 11.

После выполнения еще одного такого же оператора значение переменой изменяется и становится равным 12. Как это происходит, мы рассмотрим далее в этой главе.

Листинг 2.1. Программа на C++

```
#include <iostream>
using namespace std;
int main()
{
int number_of_bars;
    double one_weight, total_weight;

    cout << "Enter the number of candy bars in a package\n";
    cout << "and the weight in ounces of one candy bar.\n";
    cout << "Then press Enter.\n";
cin >> number_of_bars;
cin >> one_weight;

    total_weight = one_weight * number_of_bars;

    cout << number_of_bars << " candy bars\n";
    cout << one_weight << " ounces each\n";
    cout << "Total weight is " << total_weight
         << " ounces.\n";

    cout << "Try another brand.\n";
    cout << "Enter the number of candy bars in a package\n";
    cout << "and the weight in ounces of one candy bar.\n";
    cout << "Then press Enter.\n";
cin >> number_of_bars;
cin >> one_weight;

    total_weight = one_weight * number_of_bars;

    cout << number_of_bars << " candy bars\n";
    cout << one_weight << " ounces each\n";
    cout << "Total weight is " << total_weight
         << " ounces.\n";

    cout << "Perhaps an apple would be healthier.\n";
    return 0;
}
```

Пример диалога

```
Enter the number of candy bars in a package
and the weight in ounces of one candy bar.
Then press Enter.
11 2.1
11 candy bars
2.1 ounces each
Total weight is 23.1 ounces.
Try another brand.
Enter the number of candy bars in a package
and the weight in ounces of one candy bar.
Then press Enter.
12 1.8
12 candy bars
1.8 ounces each
Total weight is 21.6 ounces.
Perhaps an apple would be healthier.
```

Конечно, сравнение переменных со школьной доской является образным. Физически они представлены ячейками памяти — компилятор связывает с именем каждой переменной адрес начальной ячейки области памяти, выделенной для хранения этой переменной (о хранении переменных в памяти рассказывалось в главе 1). В такой области памяти хранится значение переменой, закодированное последовательностью нулей и единиц. Например, трем переменным в рассматриваемой программе могут быть присвоены адреса памяти 1001, 1003 и 1007. Точные числа будут зависеть от вашего компьютера, используемого компилятора и множества других факторов. Причем программист не знает, какие адреса компилятор выбирает для переменных программы, да это и не важно. Можно считать, что области памяти просто помечены именами переменных.

Что делать, если программа не выполняется

Если вы не можете добиться того, чтобы программа на C++ была откомпилирована, запущена и правильно выполнена, обратитесь к подразделу «Совет программисту: запуск программы» раздела 1.3 главы 1. Там приведены советы для различных версий компиляторов C++ и разных окружений.

Имена и идентификаторы

При изучении примеров программ вы, наверняка, обратили внимание на то, что имена переменных в них длиннее имен обычных переменных, используемых в математике. В программировании для облегчения понимания кода переменным следует давать информативные имена. Имя переменной (или другого определяемого в программе элемента) называется *идентификатором*. Идентификатор представляет собой последовательность символов произвольной длины, которая содержит буквы, цифры и символы подчеркивания и начинается либо буквой, либо символом подчеркивания. Например, имена

```
x, x1, x_1, _abc, ABC123z7, sum, RATE, count, data2, Big_Bonus
```

являются допустимыми идентификаторами, но первые пять выбраны неудачно, поскольку не несут никакой информации о назначении соответствующих программных элементов. Приведенные ниже имена не являются допустимыми идентификаторами и не будут приняты компилятором.

```
12, 3X, %change, data-1, myfirst.c, PROG.CPP
```

Первые три не подходят, поскольку начинаются не с буквы или символа подчеркивания, а три последних содержат недопустимые символы, отличные от букв, цифр и символа подчеркивания.

Язык C++ чувствителен к регистру, то есть в нем различаются прописные и строчные буквы, используемые в идентификаторах. Поэтому следующие три идентификатора воспринимаются C++ как различные и могут применяться для обозначения трех разных переменных:

```
rate, RATE, Rate
```

Однако использовать переменные с такими именами в одной программе не стоит, поскольку это может привести к путанице. Часто имена переменных в C++ задаются в нижнем регистре, хотя язык этого и не требует. Но предопределенные идентификаторы, такие как `main`, `cin` и `cout`, обязательно должны быть набраны строчными буквами. Об особенностях использования идентификаторов, заданных в верхнем регистре, мы расскажем далее в этой главе.

Идентификатор в C++ может иметь любую длину, хотя существуют компиляторы, которые игнорируют символы идентификатора, введенные сверх некоторого максимально допустимого количества.

Идентификаторы

В программах на C++ идентификаторы используются в качестве имен переменных, а также имен других элементов. Идентификатор представляет собой последовательность символов произвольной длины, которая содержит буквы, цифры и символы подчеркивания и начинается либо буквой, либо символом подчеркивания.

Существует особый класс идентификаторов, называемых *ключевыми* или *зарезервированными словами*. Эти идентификаторы имеют в C++ заранее определенное значение и не могут использоваться в качестве имен переменных или как-либо еще. Полный список ключевых слов приведен в приложении 1.

У вас может возникнуть вопрос, почему другие слова, которые мы определили как часть языка C++, не включены в этот список. Например, чем объяснить, что в нем отсутствуют слова `cin` и `cout`? Дело в том, что программисту разрешено их переопределять, хотя это и может привести к путанице. Такие предопределенные слова не являются ключевыми, однако они определены в библиотеках, требуемых стандартом языка C++. О библиотеках мы поговорим далее в этой книге, пока же пусть этот вопрос вас не беспокоит. Но учтите, что любое применение предопределенного идентификатора, не соответствующее его стандартному назначению, нежелательно, поскольку приводит к недоразумениям и может послужить причиной

ошибок в программе. Безопасней и проще всего использовать предопределенные идентификаторы таким же образом, как ключевые слова.

Объявление переменных

Каждая переменная в программе на C++ должна быть *объявлена*. Так вы указываете компилятору, а следовательно, и компьютеру, какие данные будут храниться в этой переменной. Например, следующие два объявления из программы в листинге 2.1 определяют три используемые в ней переменные:

```
int number_of_bars;
double one_weight, total_weight;
```

Когда в одном объявлении определяется несколько переменных, их имена разделяются запятыми. Обратите внимание, что каждое объявление оканчивается точкой с запятой.

Слово `int`, используемое в первой строке, является сокращением от англ. integer — целый. Учтите, что в программах на C++ нужно употреблять именно сокращенную форму. В первой строке указывается, что идентификатор `number_of_bars` будет служить именем переменной типа `int`, то есть значением этой переменной может быть только целое число, например 1, 2, –1, 0, 37 или –288.

Во второй строке два идентификатора — `one_weight` и `total_weight` — объявляются как имена переменных типа `double`. В переменных этого типа могут храниться числа с дробной частью, скажем, 1,75 или –0,55. Тип хранящегося в переменной значения именуется ее *типом данных*, а обозначение этого типа, такое как `int` или `double`, называется *именем типа данных*.

Объявление переменной

Перед использованием переменная обязательно должна быть объявлена.

Синтаксис

имя_типа имя_переменной_1, имя_переменной_2, ...;

Пример

```
int count, number_of_dragons, number_of_trolls;
double distance;
```

Каждая переменная, которая будет применяться в программе на C++, обязательно должна быть объявлена. Объявление переменной лучше всего располагать либо непосредственно перед ее использованием, либо в начале основной части программы сразу после строк

```
int main()
{
```

Делайте все, чтобы код был максимально понятным.

Объявления переменных предоставляют компилятору информацию, необходимую для их создания. Напомним, что компилятор реализует переменные как области

памяти, в которых хранятся значения этих переменных, закодированные в виде последовательности нулей и единиц. Для хранения переменных различного типа требуются неодинаковые по объему области памяти и разные способы кодирования значений. Например, отличается кодирование числа с дробной частью и без нее. Для представления букв тоже используется свой способ кодирования. Объявление переменной указывает компилятору (и следовательно, компьютеру), какой объем памяти нужно для нее выделить и как представить ее значения в виде последовательности нулей и единиц.

Синтаксис

Синтаксис языка программирования, как и любого другого языка, — это набор грамматических правил. Так, говоря о синтаксисе объявления переменной, мы имеем в виду правила составления такого объявления. Одно из условий принятия программы компилятором — ее полное соответствие синтаксическим правилам языка, однако это не обязательно означает, что программа будет делать то, что требуется.

Операторы присваивания

Самый простой способ изменить значение переменой — воспользоваться *оператором присваивания*. Такой оператор указывает компьютеру, что этой переменной нужно присвоить определенное значение. В качестве примера рассмотрим следующую строку из листинга 2.1:

```
total_weight = one_weight * number_of_bars;
```

Данный оператор говорит компьютеру, что переменной `total_weight` следует присвоить значение, равное значению переменной `one_weight`, умноженному на значение переменной `number_of_bars`. (В главе 1 мы говорили, что символ * в C++ соответствует операции умножения.)

Оператор присваивания всегда оканчивается точкой с запятой и состоит из двух частей: переменной, расположенной слева от знака равенства, и выражения, находящегося справа от него. Таким выражением может быть переменная, число или более сложная конструкция, включающая переменные, числа и знаки, обозначающие арифметические операции (например, * и +). Оператор присваивания указывает компьютеру вычислить значение выражения и присвоить результат переменной, находящейся слева от знака равенства. Далее приводится ряд примеров, которые помогут вам лучше понять, как действует этот оператор.

Вместо умножения можно использовать любую другую арифметическую операцию. Так, допустимо следующее присваивание:

```
total_weight = one_weight + number_of_bars;
```

Это такой же оператор, как в нашей программе, с той только разницей, что вместо умножения он выполняет сложение, присваивая переменной `total_weight` сумму значений переменных `one_weight` и `number_of_bars`. Конечно, если внести такое изменение в программу из листинга 2.1, она выдаст логически неверный результат, но тем не менее, будет работать.

В операторе присваивания выражение справа от знака равенства может быть просто еще одной переменной. Рассмотрим оператор

```
total_weight = one_weight;
```

который присваивает переменной total_weight значение переменной one_weight. Если включить его в текст нашей программы, результат ее выполнения получится очень маленьким (если в пакете находится более одной конфеты), но в другой программе использование такого оператора может быть вполне оправданно.

А вот еще один пример — оператор, присваивающий переменной number_of_bars значение 37:

```
number_of_bars = 37;
```

Непосредственно заданное в программе число, такое как 37 в нашем примере, называется *константой*, поскольку в отличие от переменной это значение не может измениться.

Еще более интересно, что с двух сторон от знака равенства может находиться одна и та же переменная. Например:

```
number_of_bars = number_of_bars + 3;
```

На первый взгляд оператор может показаться странным, поскольку с точки зрения математики данное выражение означает, что переменная number_of_bars равна самой себе плюс три. Однако в программировании этот оператор читается так: «сделать новым значением переменной number_of_bars ее старое значение плюс три». Знак равенства в C++ используется не так, как в математике.

Операторы присваивания

При выполнении оператора присваивания сначала вычисляется выражение справа от знака равенства, а затем результат присваивается переменной, указанной слева от него.

Синтаксис

переменная = выражение;

Пример

```
distance = rate * time;
count = count + 2;
```

Ловушка: неинициализированные переменные

Переменная не имеет смыслового значения до тех пор, пока оно не будет присвоено ей программой. Так, если переменной minimum_number с помощью оператора присваивания или иным путем (например, используя поток ввода cin) еще не присвоено значение, следующий оператор будет ошибочным:

```
desired_number = minimum_number + 10;
```

Поскольку minimum_number пока не имеет смыслового значения, его не имеет и все выражение справа от знака равенства. Переменная, которой еще не присвоено значение, называется *неинициализированной*, но это не означает, что она не содержит

никакого значения. То, что в ней хранится до инициализации, программисты называют «мусором». Значение неинициализированной переменной — это просто некая последовательность нулей и единиц, оставшаяся в памяти от предыдущей программы, использовавшей эту область. При повторном запуске программы та же неинициализированная переменная будет, скорее всего, содержать уже другую последовательность нулей и единиц. Вот вам практическая подсказка: если при использовании одних и тех же входных данных программа каждый раз выдает различные выходные данные, можно предположить, что в этом виновата неинициализированная переменная.

Во избежание случайного использования неинициализированных переменных лучше инициализировать все переменные сразу при их объявлении. Для этого в объявление переменой нужно добавить знак равенства и начальное значение, как показано ниже:

```
int minimum_number = 3;
```

Этот оператор объявляет переменную `minimum_number` типа `int` и присваивает ей значение 3. Чаще всего переменные инициализируются просто константами, хотя можно использовать и более сложные выражения с применением таких операций, как сложение или умножение. В объявлении, включающем несколько переменных, можно инициализировать любое их количество: все, часть переменных или ни одну из них. Например, в приведенной далее строке объявляются три переменные и инициализируются две из них:

```
double rate = 0.07, time, balance = 0.0;
```

Язык C++ поддерживает альтернативный синтаксис инициализации переменных при объявлении. Так, строка

```
double rate(0.07), time, balance(0.0);
```

эквивалентна предыдущей.

В одних случаях переменную лучше инициализировать при объявлении, а в других — позднее, все зависит от логики конкретной программы.

Инициализация переменных при объявлении

Переменную можно инициализировать, то есть присвоить ей начальное значение, прямо при ее объявлении.

Синтаксис

```
имя_типа имя_переменной_1 = выражение_1,
         имя_переменной_2 = выражение_2, ...;
```

Пример

```
int count = 0, limit = 10, fudge_factor = 2;
double distance = 999.99;
```

Альтернативный синтаксис

```
имя_типа имя_переменной_1(выражение_1),
         имя_переменной_2(выражение_2), ...;
```

Пример

```
int count(0), limit(10), fudge_factor(2);
double distance(999.99);
```

Совет программисту: используйте информативные имена

Имена переменных и другие имена, используемые в программе, должны нести информацию о смысле или назначении соответствующих им элементов. Это делает программу более понятной. Для примера сравните следующие две строки:

```
x = y * z;
```

и

```
distance = speed * time;
```

Два приведенных здесь оператора выполняют одно и то же действие, но если о первом ясно только, что он перемножает два каких-то числа и заносит результат в переменную, то смысл второго совершенно понятен: он умножает скорость на время и заносит результат в переменную, обозначающую расстояние.

Упражнения для самопроверки

1. Запишите объявления двух переменных с именами `feet` и `inches`. Обе они должны иметь тип `int` и в объявлениях инициализироваться нулями. Выполните упражнение в двух вариантах, используя альтернативный синтаксис инициализации при объявлении.
2. Запишите объявления двух переменных с именами `count` и `distance`. Переменная `count` должна иметь тип `int` и инициализироваться значением `0`, а переменная `distance` типа `double` — значением `1.5`.
3. Запишите оператор, помещающий в переменную `sum` сумму значений переменных `n1` и `n2`, которые должны иметь тип `int`.
4. Запишите оператор, увеличивающий значение переменной `length` типа `double` на 8,3.
5. Запишите оператор, помещающий в переменную `product` ее исходное значение, умноженное на значение переменной `n`. Переменные должны иметь тип `int`.
6. Напишите программу, содержащую операторы, которые выводят значения пяти или шести переменных, объявленных, но не инициализированных. Откомпилируйте и запустите эту программу. Объясните, что она выводит.
7. Запишите подходящие имена для следующих переменных:
   а) переменная для хранения скорости автомобиля;
   б) переменная для хранения оклада служащего;
   в) переменная для хранения максимальной экзаменационной оценки.

2.2. Ввод-вывод

Мусор введешь — мусор и выведешь.

Программистская пословица

В программах на C++ можно использовать разные способы выполнения ввода и вывода. Здесь мы опишем так называемый потоковый ввод-вывод. *Поток ввода* —

это просто поток входных данных, направляемый в компьютер для использования программой. Слово «поток» означает, что программа обрабатывает входные данные независимо от источника их поступления. При работе с примерами из этого раздела предполагается, что данные вводятся с клавиатуры. О том, как программа считывает входные данные из файла, мы расскажем в главе 5, и вы увидите, что одни и те же операторы ввода используются для ввода данных как из файла, так и с клавиатуры. Аналогично *поток вывода* — это поток генерируемых программой выходных данных. В этом разделе мы будем считать, что выходные данные направляются на экран терминала, а в главе 5 расскажем, как вывести данные в файл.

Вывод с помощью потока cout

Значения переменных и текстовые строки в C++ можно отобразить на экране с помощью стандартного потока вывода cout (выходные данные могут быть представлены произвольной комбинацией переменных и строк). Рассмотрим следующую строку из программы, показанной в листинге 2.1:

```
cout << number_of_bars << " candy bars\n";
```

Данный оператор указывает компьютеру вывести два элемента: значение переменной number_of_bars и строку «candy bars», а затем переместить курсор в начало следующей строки (это указывается непечатаемым сочетанием символов \n). Обратите внимание, что слово cout не требуется для каждого выводимого элемента. Достаточно задать его в начале строки, а затем указать все выводимые элементы, предварив каждый из них символом <<. Символ < — это обыкновенный символ «меньше». Два этих символа, расположенных подряд без пробела между ними, называются *оператором вывода*. Весь оператор, начинающийся с cout, оканчивается точкой с запятой.

Приведенный выше оператор эквивалентен следующим двум:

```
cout << number_of_bars;
cout << " candy bars\n";
```

Используя поток cout, можно вывести результат вычисления арифметического выражения, как в следующем примере:

```
cout << " The total cost is $" << (price + tax);
```

Здесь price и tax являются переменными.

Поскольку некоторые компиляторы требуют заключения такого арифметического выражения в скобки, лучше использовать скобки всегда.

Два идущих подряд оператора, начинающихся с cout, всегда можно объединить в один. Рассмотрим в качестве примера следующий фрагмент программы, приведенной в листинге 2.1:

```
cout << number_of_bars << " candy bars\n";
cout << one_weight << " ounces each\n";
```

Данные две строки можно переписать следующим образом:

```
cout << number_of_bars << " candy bars\n" << one_weight
<< " ounces each\n";
```

При этом результат никак не изменится.

Если вы хотите, чтобы строки программы умещались на экране, то длинные операторы, начинающиеся с cout, можно разбивать на две и более строк. Лучше всего записывать их так:

```
cout << number_of_bars << " candy bars\n"
     << one_weight << " ounces each\n";
```

Строку в кавычках нельзя разбивать на две строки, но можно переходить на новую строку в любом месте, где в исходной строке допустимо использование пробела. Компьютером принимается какое угодно расположение отступов и разрывов строк, однако лучше всего следовать образцу программ, приводимых в этой книге. Обратите внимание, что каждый оператор, производящий вывод при помощи потока cout, содержит единственную (завершающую) точку с запятой независимо от того, на сколько строк он разбит.

Особое внимание следует уделять заключенным в кавычки выводимым строкам. Заметьте, что они должны быть двойными, представленными одним символом двойной кавычки (имеющимся на клавиатуре), а не двумя подряд идущими символами одинарной. Также учтите, что начало и конец строки отмечает один и тот же символ: в данном случае используются «прямые» кавычки, а не парные символы правой и левой.

Теперь поговорим о пробелах внутри кавычек. Компьютер не вставляет их ни до, ни после элементов, выводимых потоком cout, так что эти элементы выводятся вплотную один за другим. Поэтому выходные строки в наших примерах часто начинаются и/или оканчиваются пробелами, которые предназначены для разделения выводимых элементов. Если же между выводом двух выражений или переменных не выводится символьная строка, то для их разделения нужно добавить еще один элемент: вывод строки, состоящей из единственного пробела, как в следующем примере:

```
cout << first_number << " " << second_number;
```

Как мы уже говорили в главе 1, символ \n указывает компьютеру начать новую строку. Если не записывать этот символ в выходной поток, все выходные данные выведутся в одной строке. В соответствии с настройкой экрана результат будет либо, не помещаясь, выходить за границы экрана, либо произвольно разбиваться на строки так, чтобы отображалась вся выводимая информация. В C++ перевод строки считается отдельным символом, который вставляется в выводимую строку с помощью специального обозначения \n (без пробела), всегда задаваемого в кавычках. Символ, который записывается в выходной поток с помощью этого обозначения, называется *символом новой строки*.

Директивы include и пространства имен

До сих пор все наши программы начинались такими строками:

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

Первая из них называется *директивой* include. Она включает в программу библиотеку iostream, для того чтобы в этой программе можно было применять идентификаторы cin и cout.

Названные идентификаторы определены в файле iostream, и приведенная выше директива эквивалентна копированию в программу данного файла.

Вторую строку объяснить несколько сложнее. В C++ все имена разделяются на пространства имен. *Пространство имен* — это некоторый набор имен, подобных cin и cout. Используемое в программе пространство имен задается оператором

```
using namespace std;
```

называемым директивой using. Данная директива указывает, что в программе применяется пространство имен std (сокращение от англ. standard — стандартный), то есть все употребляемые в программе имена будут иметь значения, определенные в этом пространстве имен. В нашем примере идентификаторы cin и cout определены в файле iostream, где сказано, что они относятся к пространству имен std. Поэтому для их применения нужно указать компилятору, что в программе используется пространство имен std.

На данном этапе вам достаточно этих знаний о пространствах имен. Но для чего вообще нужны пространства имен? Дело в том, что в программах на языке C++ используется такое количество всевозможных имен, что нередко одно и то же имя носят два совершенно разных элемента, иными словами, одно и то же имя может иметь два разных определения. Для того чтобы ссылки на такие элементы в программах были однозначными, C++ разделяет элементы на группы, в каждой их которых имена элементов не повторяются.

Однако поймите, что пространство имен — не просто группа имен. Это программный код на языке C++, определяющий значения имен, то есть набор объявлений и определений. Все спецификации имен C++ разделяются на группы (называемые пространствами имен), чтобы каждое имя в пространстве имен имело единственную спецификацию (определение).

Ну а как быть, если вам потребуется использовать два одноименных элемента из разных пространств имен? Это возможно и достаточно просто, но о том, как это сделать, мы расскажем позднее.

Некоторые версии языка C++ используют более старую форму директивы include:

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

В случае, когда ваша программа не компилируется или не запускается со строками

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

попробуйте заменить их более старой формой. И если после этого программа запустится, значит, у вас старый компилятор C++, который следует заменить новой версией.

Управляющие последовательности

Символ обратной косой черты (\) говорит компилятору, что следующий за ним символ нужно интерпретировать не так, как он интерпретируется сам по себе,

а как некоторый управляющий код. Пара идущих подряд символов, в которой первым является символ \, называется *управляющей последовательностью*. В C++ определено несколько управляющих последовательностей.

Если вам потребуется включить в строковую константу символ \ или " (двойную кавычку), перед ними нужно поставить символ обратной косой черты. В первом случае компилятор понимает, что следующий символ нужно интерпретировать не как управляющий, а просто как обратную косую, входящую в состав строки, а во втором случае — что следующая за ним двойная кавычка не завершает строку, а входит в ее состав.

Обратная косая без последующего символа, вместе с которым она образует управляющую последовательность, разными компиляторами интерпретируется по-разному. Например, встретив последовательность символов \z, один компилятор воспримет ее просто как символ z, а другой выдаст сообщение об ошибке. В стандарте ANSI сказано, что поведение управляющих последовательностей, не поддерживающихся этим стандартом, заранее не определено. Это означает, что компилятор будет действовать так, как было задумано его автором. Поэтому код, где встречаются такие управляющие последовательности, нельзя использовать, так что лучше их не применять. Вот несколько поддерживаемых управляющих последовательностей:

Новая строка	`\n`
Символ табуляции	`\t`
Звуковой сигнал	`\a`
Обратная косая	`\\`
Двойная кавычка	`\"`

Если при выводе данных вы захотите перейти на новую строку, это можно сделать так:

```
cout << "\n";
```

Еще один способ — воспользоваться идентификатором `endl`, представляющим строку `"\n"`:

```
cout << endl;
```

Хотя `\n` и `endl` означают одно и то же, они используются по-разному. Символы `\n` всегда должны быть заключены в кавычки, тогда как `endl` задается без них.

Запомните хорошее правило для выбора между `\n` и `endl`: если вы задаете заключенную в кавычки строку, после которой необходим перевод строки, добавьте в ее конец `\n`:

```
cout << "Fuel efficiency is "
     << mpg << " milles per gallon\n";
```

Если же перевод строки выполняется после вывода переменной или выражения, используйте `endl`:

```
cout << "You entered "
     << number << endl;
```

Как начать новую строку в выходном потоке

Для того чтобы начать новую строку, можно включить в строку выводимого текста, заключенную в кавычки, символы \n:

```
cout << "You have definitely won\n"
     << " one of the following prizes:\n";
```

Напомним, что символы \n вводятся подряд без пробела между ними.

В качестве альтернативы можно начать новую строку с помощью идентификатора endl. Вот оператор, эквивалентный предыдущему:

```
cout << "You have definitely won" << endl
     << " one of the following prizes:" << endl;
```

Совет программисту: заканчивайте каждую программу выводом символа новой строки

Существует хорошее правило: перед завершением каждой программы выводить символ новой строки. Если последний выводимый программой элемент является строкой, в ее конец можно добавить \n, а если нет, то последним в программе оператором вывода нужно вывести endl. Это действие позволит избежать двух проблем. Первая состоит в том, что некоторые компиляторы не позволяют отобразить на экране последнюю выводимую программой строку до тех пор, пока не будет выведен символ новой строки. Вторая же проблема связана с тем, что при запуске следующей программы ее выходные данные могут оказаться в одной строке с последними данными, выведенными предыдущей программой. В любом случае вывод символа новой строки в конце программы делает ее более переносимой.

Форматирование чисел с дробной частью

Когда компьютер выводит значение типа double, его формат может оказаться не таким, как требуется. Например, оператор

```
cout << "The price is $" << price << endl;
```

может выводить разные результаты.

Если переменная price содержит значение 78.5, результат может быть таким:

```
The price is $78.500000
```

или таким:

```
The price is $78.5
```

или даже таким (об этом формате рассказывается в разделе 2.3):

```
The price is $7.850000e01
```

Но маловероятно, что он будет следующим:

```
The price is $78.50
```

хотя именно этот формат имеет смысл, так как выводится денежная сумма.

Для того чтобы результат имел требуемую форму, программа должна содержать определенные инструкции, указывающие компьютеру, как выводить числа.

Существует «магическая формула», которую можно вставить в программу, чтобы числа, содержащие десятичную точку (такие, как числа типа double), выводились с нужным количеством десятичных цифр. Так, чтобы после десятичной точки выводились две десятичные цифры, нужно выполнить следующие операторы:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

Если вставить эти три оператора в программу, все выполняемые после них операторы вывода, использующие поток cout, будут выводить значения типа double с двумя цифрами после точки. Предположим, что оператор

```
cout << "Цена $" << price << endl;
```

выполняется после приведенных трех операторов, и переменная price содержит значение 78.5. Тогда результат будет таким:

```
Цена $78.50
```

Вместо значения 2 можно задать любое другое неотрицательное целое число или даже переменную типа int.

О последовательности операторов, устанавливающей количество знаков после точки, подробно рассказывается в главе 5. Пока же можете считать ее одной длинной инструкцией, указывающей компьютеру, как должны выводиться числа, содержащие дробную часть.

Для изменения количества знаков после десятичной точки, чтобы различные значения в программе выводились по-разному, можно повторить эту же последовательность операторов, но с другим значением. Причем на этот раз достаточно будет воспроизвести только ее последнюю строку. Если вышеприведенная комбинация операторов один раз уже встречалась в программе, строка

```
cout.precision(5);
```

изменяет количество знаков после десятичной точки на 5 для всех выводимых далее значений типа double.

Вывод значений типа double

При вставке в программу следующей «магической формулы»:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

все числа типа double (и других типов, включающих дробную часть) будут выводиться в формате с двумя цифрами после десятичной точки. Вместо значения 2 можно задать другое количество десятичных знаков или даже переменную типа int.

Ввод с помощью потока cin

Ввод с помощью стандартного потока ввода cin аналогичен выводу с помощью потока вывода cout. Их синтаксис различается только тем, что при работе с потоком

cout используется оператор <<, а при работе с потоком cin — оператор >>. Так, в программе из листинга 2.1 переменные number_of_bars и one_weight заполняются с помощью следующих операторов, начинающихся словом cin (приведенных вместе с операторами, выводящими указания пользователю):

```
cout << "Enter the number of candy bars in a package\n";
cout << "and the weight in ounces of one candy bar.\n";
cout << "Then press Enter.\n";
cin >> number_of_bars;
cin >> one_weight;
```

В одном операторе ввода при помощи потока cin может быть перечислено более одной переменной. Например, следующие строки

```
cout << "Enter the number of candy bars in a package\n";
cout << "and the weight in ounces of one candy bar.\n";
cout << "Then press Enter.\n";
cin >> number_of_bars >> one_weight;
```

эквивалентны предыдущим.

Можно разбить оператор ввода при помощи потока cin на две строки:

```
cin >> number_of_bars
    >> one_weight;
```

Обратите внимание на то, что так же как для оператора вывода при помощи cout, для оператора ввода при помощи cin необходима только одна точка с запятой — в конце.

Когда программа достигает оператора, начинающегося словом cin, она ждет ввода значений с клавиатуры, а затем устанавливает первую переменную равной первому введенному с клавиатуры значению, вторую переменную — равной второму значению и т. д. Программа не считывает введенные данные, пока не будет нажата клавиша Enter, что дает возможность пользователю возвращаться назад и исправлять ошибки, допущенные при вводе значения.

Вводимые числа должны разделяться одним или более пробелами или символом перехода на новую строку. Если, например, вы захотите ввести числа 12 и 5 и не поставите между ними пробела, они будут восприняты как одно число 125. Выполняя ввод при помощи потока cin, компьютер пропускает любое количество пробелов и символов перехода на новую строку во входном потоке, пока не встретит следующее входное значение. Таким образом, количество пробелов и символов перехода на новую строку, разделяющих входные значения, не важно.

Поток ввода cin

С помощью стандартного потока ввода cin можно помещать в заданные переменные значения, введенные с клавиатуры.

Синтаксис

cin >> *переменная_1* >> *переменная_2* >> ...;

Примеры

```
cin >> number >> size;
cin >> time_to_go
    >> points_needed;
```

Программирование ввода и вывода

Ввод и вывод, или, как зачастую говорят, *ввод-вывод* — это часть программы, которую видит пользователь, поэтому она должна быть особенно тщательно спроектирована.

При выполнении ввода с использованием потока `cin` компьютер ожидает ввода с клавиатуры некоторых данных (если никакие данные не вводятся, он будет ждать до бесконечности). Поскольку компьютер не будет автоматически просить пользователя ввести информацию, это должна делать сама программа. Поэтому наша демонстрационная версия содержит такие строки:

```
cout << "Enter the number of candy bars in a package\n";
cout << "and the weight in ounces of one candy bar.\n";
cout << "Then press Enter.\n";
```

Здесь операторы предлагают пользователю ввести данные. Программа всегда должна запрашивать ввод, а не просто его ожидать.

Когда данные вводятся с терминала, они по мере ввода отображаются на экране. Однако программа и сама должна обязательно вывести введенные пользователем значения, чтобы тот мог убедиться, что они прочитаны программой правильно. Такой вывод называется *эхо-выводом*. Учтите, что если вводимые данные хорошо выглядят на экране, это еще не означает, что они будут правильно прочитаны компьютером. Возможны самые разные проблемы: от незамеченных пользователем опечаток до внутренних проблем компьютера или программы. Так что эхо-вывод помогает обеспечить корректность входных данных.

Совет программисту: переводы строки при вводе-выводе

Входные и выходные данные могут располагаться на одной строке, и иногда такой интерфейс получается наиболее удобным для пользователя. Чтобы вводимые данные отображались в той же строке, что и запрос, достаточно не выводить в конце запроса `\n` или `endl`. В качестве примера рассмотрим следующий код:

```
cout << "Enter a cost per person: $";
cin >> cost_per_person;
```

После выполнения первого из этих двух операторов на экран будет выведено следующее:

```
Enter a cost per person: $
```

Когда пользователь введет значение, оно отобразится в той же строке:

```
Enter a cost per person: $1.25
```

Упражнения для самопроверки

8. Запишите оператор, выводящий на экран следующее сообщение:

```
The answer to the question of
Life, the Universe, and Everything is 42.
```

9. Запишите оператор, который помещает в переменную the_number (типа int) введенное с клавиатуры число. Вставьте перед ним оператор, запрашивающий у пользователя целое число.
10. Какой оператор нужно включить в программу, чтобы числа типа double выводились с тремя цифрами после десятичной точки?
11. Напишите законченную программу на C++, выводящую на экран фразу Hello word. Больше ничего эта программа делать не должна.
12. Напишите законченную программу на C++, принимающую от пользователя два целых числа и выводящую их сумму. Обязательно запросите данные, выполните их эхо-вывод и укажите, что за результат вы выводите.
13. Запишите оператор, выводящий символ новой строки и символ табуляции.
14. Напишите короткую программу, объявляющую и инициализирующую переменные типа double с именами one, two, three, four, five и значениями 1.0, 1.414, 1.732, 2.0, 2.236. Затем напишите операторы, выводящие приведенные ниже заголовки и таблицу. Для выравнивания столбцов таблицы используйте управляющую последовательность \t. Если вы не знаете, как используется символ табуляции, поэкспериментируйте с ним при выполнении этого упражнения. Табуляция работает подобно механическому табулятору печатной машинки. После нее вывод начинается со следующего столбца, ширина которого обычно кратна восьми пробелам. Во многих редакторах и текстовых процессорах ширину табуляции можно настраивать, но в нашем случае она фиксированная. Вывод программы должен быть таким:

```
N       Square Root
1       1.000
2       1.414
3       1.732
4       2.000
5       2.236
```

2.3. Типы данных и выражения

Они никогда не будут счастливы вместе. Это не ее тип.

Случайно услышано в кафе

Типы int и double

Концептуально значения 2 и 2.0 — это одно и то же число. Однако C++ считает их значениями разных типов. Значение 2 рассматривается как целое и имеет тип int, тогда как значение 2.0 принадлежит к типу double, поскольку содержит дробную часть, хотя и равную нулю. Обратите внимание на разницу между компьютерной и обычной школьной математикой — в силу определенных особенностей хранимые в компьютере числа несколько отличаются от абстрактного определения понятия «число». Целые числа в C++ ведут себя привычным для нас образом, и тип данных

int не таит в себе никаких неожиданностей. Иное дело — значения типа double. Поскольку они хранятся с ограниченным количеством десятичных знаков, это всегда приблизительные значения. Точность значений типа double зависит от конкретной компьютерной системы, но она почти наверняка составляет не менее 14 знаков. Для большинства приложений этого достаточно, хотя даже в самых простых случаях возможны некоторые проблемы. Поэтому, если известно, что значениями некоторой переменной всегда будут целые числа из допустимого диапазона, лучше всего объявить ее как переменную типа int.

Числовые константы типа double записываются иначе, чем константы типа int. Целочисленные константы не содержат десятичной точки, тогда как константы типа double можно записывать в двух форматах. Простейшая форма константы этого типа — повседневный американский формат записи десятичных чисел, в котором целая и дробная части числа разделяются точкой. Обратите внимание: ни пробелов, ни запятых числовые константы в C++ содержать не должны, и это касается как типа int так и типа double.

Более сложную форму констант типа double часто называют *научным форматом* или *форматом с плавающей запятой*. (Напомним, что в языке C++ точка в числах представляет десятичную запятую.) Этот формат особенно удобен для записи очень больших и очень маленьких чисел. Например:

3.67×10^{17} или 367 000 000 000 000 000.0

в C++ лучше всего выразить константой 3.67e17. А число

5.89×10^{-6} или 0.00000589

константой 5.89e-6. Буква e соответствует слову экспонента (от англ. exponent), и ее использование в записи числа означает «умножить на десять в указанной степени».

Такая форма записи применяется в связи с тем, что на клавиатуре отсутствуют стандартные клавиши для набора знака возведения в степень в привычном для нас виде. Число и его знак, расположенные после буквы e, можно рассматривать как указание направления и количества знаков, на которое нужно переместить десятичную точку. Например, чтобы превратить число 3.49e4 в число без буквы e (то есть в 34900.0), следует переместить десятичную точку на четыре позиции вправо. Если же после буквы e указано отрицательное число, десятичная точка перемещается влево с добавлением соответствующего количества нулей. Так, 3.49e-4 эквивалентно числу 0.0349.

В число перед буквой e может входить десятичная точка, хотя это не обязательно. Что касается значения экспоненты (показателя степени), заданной после буквы e, то оно не должно содержать точку.

Поскольку память компьютера имеет ограниченный объем, для хранения чисел отводится конечное число байт, которое различно для разных типов чисел. Максимально допустимое число типа double всегда больше максимального числа типа int. Практически любая реализация языка C++ допускает значения типа int до 32 767 и значения типа double до примерно 10^{308}.

Почему double

Почему тип чисел с дробной частью называется double («двойной»), и имеется ли тип данных single вдвое меньшего размера? В языке C++ такой тип данных отсутствует, но есть языки, в которых он существует. Традиционно во многих языках программирования использовалось два типа данных для чисел с дробной частью. Один имел меньший размер и точность, а второй — вдвое больший размер и более высокую точность и допускал хранение больших чисел (хотя программистов обычно заботит точность, а не размер чисел). Числа первого типа назывались числами одинарной точности, а второго — числами двойной точности. Вот по этой-то традиции тип данных, который примерно соответствует числам двойной точности, в C++ был назван double. В языке C++ имеется и тип данных, соответствующий числам одинарной точности; он именуется float. Кроме того, в C++ представлен еще и третий тип чисел с дробной частью, называемый long double. Эти типы описаны в следующем разделе, но в дальнейшем в этой книге мы не будем их использовать.

Другие числовые типы

Помимо int и double в C++ имеются и другие числовые типы данных. Некоторые из них перечислены в табл. 2.1. Разные числовые типы позволяют хранить числа разного размера и разной точности (то есть с большим или меньшим количеством цифр после десятичной точки). В таблице приведены соответствующие каждому типу данных объем памяти, используемый для его хранения, диапазон значений и точность. Все они зависят от конкретной системы и у вас могут быть другими, а здесь перечислены просто для того, чтобы вы могли составить некоторое общее представление о числовых типах данных языка C++.

Хотя некоторые из этих числовых типов данных идентифицируются двумя словами, переменные таких типов объявляются точно так же, как переменные типа int или double. Например, в следующей строке объявляется переменная типа long double:

```
long double big_number;
```

Имена long и long int являются именами одного типа данных. Поэтому следующие два объявления эквивалентны:

```
long big_toatl;
long int big_total;
```

Конечно, в программе следует использовать только одно из приведенных двух объявлений переменной big_total, но какое именно — не имеет значения. Кроме того, помните, что имя типа long означает то же, что long int, но не long double.

Типы данных, предназначенные для хранения целых чисел, такие как int, называются *целочисленными типами*. Типы данных для чисел с дробной частью, например double, именуются *типами с плавающей запятой*. Свое название они получили потому, что число, записанное обычным образом (скажем, 392.123), перед записью в память преобразуется в формат, напоминающий научный (3.92123e2). Когда компьютер выполняет это преобразование, десятичная точка (запятая) перемещается в новое положение, поэтому ее и называют плавающей.

Следует иметь в виду, что в C++ существуют и другие числовые типы данных, но в этой книге мы будем использовать только типы int, double и иногда long. Для большинства простых приложений достаточно первых двух типов, однако если программа оперирует очень большими целыми числами, для них потребуется тип данных long.

Таблица 2.1. Некоторые числовые типы данных

Имя типа	Объем занимаемой памяти	Диапазон значений	Точность
short (или short int)	2 байта	От –32 768 до 32 767	Не применима
int	4 байта	От –2 147 483 648 до 2 147 483 647	Не применима
long (или long int)	4 байта	От –2 147 483 648 до 2 147 483 647	Не применима
float	4 байта	Примерно от 10^{-38} до 10^{38}	7 цифр
double	8 байт	Примерно от 10^{-308} до 10^{308}	15 цифр
long double	10 байт	Примерно от 10^{-4932} до 10^{4932}	19 цифр

Это только примерные значения, демонстрирующие различия между типами данных. В вашей системе они могут быть другими. *Точность* означает количество значащих цифр, включая и цифры до десятичной точки. Диапазоны значений типов float, double и long double приведены для положительных чисел. Отрицательные числа имеют те же диапазоны, но со знаком минус.

Тип данных char

До сих пор мы говорили только о числовых типах данных, и возможно, у вас сложилось неверное впечатление, что компьютеры и язык C++ используются исключительно для числовых вычислений. Теперь пришло время познакомиться и с другими типами данных. Первый из них – тип char (сокращение от англ. character – символ). Значениями этого типа являются одиночные символы, такие как буква, цифра или знак препинания. Вот пример объявления двух переменных типа char с именами symbol и letter:

```
char symbol, letter;
```

В переменной типа char может храниться один символ, соответствующий клавише, нажатой на клавиатуре. Так, это может быть символ 'A' или '+', или 'a'. Учтите, что одна и та же буква, набранная в верхнем и нижнем регистре, рассматривается как разные символы.

Существует тип данных для строк, содержащих более одного символа. Мы расскажем о нем позднее, хотя вы с ним уже немного знакомы, ведь мы не раз задавали в операторах вывода, использующих поток cout, строки, заключенные в двойные

кавычки. Это и были значения строкового типа. Примером может служить строка из листинга 2.1:

```
"Enter the number of candy bars in a package\n";
```

Обратите внимание, что такие строковые константы всегда заключаются в двойные кавычки, тогда как символьные константы задаются в одинарных кавычках. Таким образом, 'A' и "A" в программе имеют разное значение. Если 'A' — значение типа char, которое может храниться в переменной типа char, то "A" является строкой символов, и для ее хранения требуется переменная строкового типа. Тот факт, что строка содержит только один символ, еще не делает ее значением типа char. Учтите также, что и в символьных и в строковых константах открывающей и закрывающей кавычкой служит один и тот же символ.

Применение типа данных char показано в программе, приведенной в листинге 2.2. Эта программа просит пользователя ввести два инициала без точек, а затем выводит их сначала подряд без пробела, а затем еще раз с пробелом между ними. Заметьте, что пользователь вводит между первым и вторым инициалами пробел, а программа пропускает его и считывает букву B как второй введенный символ. Когда для чтения данных в переменные типа char используется поток ввода cin, компьютер пропускает все пробелы и символы перевода строки до тех пор, пока не встретит непустой символ, который и считывается в переменную. Поэтому не имеет значения, присутствуют во введенных данных пробелы или нет. Программа из листинга 2.2 выдаст одинаковый результат независимо от того, введет ли пользователь пробел между инициалами, как в примере диалога, или наберет их подряд, JB.

Листинг 2.2. Тип данных char

```
#include <iostream>
using namespace std;
int main()
{
char symbol1, symbol2, symbol3;

    cout << "Enter two initials, without any periods:\n";
    cin >> symbol1 >> symbol2;

    cout << "The two initials are:\n";
    cout << symbol1 << symbol2 << endl;

    cout << "Once more with a space:\n";
    symbol3 = ' ';
    cout << symbol1 << symbol3 << symbol2 << endl;

    cout << "That's all.";

    return 0;
}
```

Пример диалога

```
Enter two initials, without any periods:
J B
```

```
The two initials are:
JB
Once more with a space:
J B
That's all.
```

Тип данных bool

Последний тип данных, который мы представим в этой главе, называется bool. Он сравнительно недавно добавлен в язык C++ комитетом ISO/ANSI (International Organization for Standardization/American National Standards Institute). Выражения типа bool называют *логическими* или *булевыми* по имени английского математика Джорджа Буля (1815–1864), сформулировавшего правила математической логики.

Результатом вычисления логического выражения может быть только одно из двух значений, true («истина») или false («ложь»). Логические выражения используются в операторах ветвления и цикла, которые рассмотрены в разделе 2.4. Там же подробнее рассказывается и о самом типе данных bool.

Совместимость типов данных

Как правило, значения одного типа данных нельзя сохранять в переменных других типов. Так, большинство компиляторов отвергнет следующее присваивание:

```
int int_variable;
int_variable = 2.99;
```

Причина в несоответствии типов. Константа 2.99 имеет тип double, а переменная int_variable — тип int. К сожалению, не все компиляторы одинаково отреагируют на приведенный код. Одни из них выдадут сообщение об ошибке, другие предупреждающее сообщение, а некоторые и вовсе не станут против него возражать. Но даже если компилятор позволит выполнить приведенное выше присваивание, он, скорее всего, присвоит переменной int_variable значение типа int, причем даже не 3, наиболее близкое к 2.99, а 2. (Поскольку нельзя рассчитывать, что компьютер примет подобное присваивание значения типа double переменной типа int, лучше его никогда не выполнять.)

То же самое произойдет и в том случае, если вместо числа 2.99 в программе будет использоваться переменная типа double. Например, большинство компиляторов не примет код

```
int int_variable;
double double_variable;
double_variable = 2.00;
int_variable = double_variable;
```

Тот факт, что число 2.00 с нашей точки зрения является целочисленным, не имеет никакого значения. В программе оно относится к типу double, а не int. И даже если заменить в приведенном выше фрагменте 2.00 числом 2, ничего не изменится. Переменные int_variable и double_variable имеют разные типы, и именно в этом заключается проблема.

Даже если компилятор позволяет смешивать типы данных в операторах присваивания, в большинстве случаев делать это не рекомендуется, поскольку из-за такого смешения программа становится менее переносимой, а ее понимание затрудняется. Так, если компилятор позволит присвоить переменной типа int значение 2.99, в ней окажется число 2, а не 2.99, что из программного кода не очевидно. Это запутывает пользователя, он может подумать, что в переменной находится число 2.99, или же просто не поймет, какое значение в ней содержится.

Существует несколько случаев, когда разрешается присваивать значение одного типа переменной другого типа. Например, можно присвоить значение типа int переменной типа double. Вот вполне допустимый код:

```
double double_variable;
double_variable = 2;
```

В данном случае в переменную double_variable помещается значение 2.0.

В переменную типа char можно поместить значение типа int (например, 65), и наоборот, в переменную типа int можно поместить значение типа char (скажем, 'Z'), хотя обычно так поступать не рекомендуется. C++ может рассматривать символы как целые числа, эта способность унаследована им от языка C, в котором она использовалась для разных целей. Переменные типа char занимают меньше памяти, чем переменные типа int, поэтому выполнение арифметических операций над первыми позволяет сэкономить память. Но такая программа малопонятна, и лучше использовать каждый тип данных по его прямому назначению.

Общее правило работы с переменными таково: значение одного типа нельзя присваивать переменной другого типа. Из этого правила имеется множество исключений, но даже если компилятор не требует его строгого соблюдения, нарушать правило не стоит. Помещение данных одного типа в переменную другого типа может служить источником разных проблем, так как значение должно быть преобразовано к типу переменной, и результат может не соответствовать вашим ожиданиям.

Значения типа bool можно присваивать переменным целочисленных типов (short, int и long), а целые числа можно присваивать переменным типа bool. Однако, как и для других типов данных, лучше этого не делать. Для полноты изложения, а также для того, чтобы вы могли прочитать и понять любую программу, все же уточним: когда целое число присваивается переменной типа bool, 0 преобразуется в false, а любое ненулевое значение в true. Если значение типа bool присваивается переменной целочисленного типа, значение true преобразуется в 1, а false в 0.

Арифметические операторы и выражения

В программе на языке C++ можно объединять переменные и числа с помощью арифметических операторов сложения (+), вычитания (-), умножения (*) и деления (/). Например, в следующем операторе присваивания из программы листинга 2.1 с помощью оператора * перемножаются числа, хранящиеся в двух переменных. (Результат помещается в переменную, заданную слева от знака равенства.)

```
total_weight = one_weight * number_of_bars;
```

Все арифметические операторы могут использоваться с числами типа int, типа double и даже с числами различающихся между собой типов. Тип результирующего значения зависит от типов операндов (то есть участвующих в операции чисел). Если оба они имеют тип int, результат тоже будет иметь тип int. Если один или оба операнда имеют тип double, получается результат типа double. Так, если переменные base_amount и increase имеют тип int, результат выражения

```
base_amount + increase
```

тоже будет иметь тип int. Однако если бы одна или обе эти переменные имели тип double, результат имел бы тип double. Это верно также для операторов –, * и /.

Тип результата может оказаться для программы более важным, чем вы предполагаете. Например, в выражении 7.0/2 операнд 7.0 имеет тип double, поэтому результат имеет тип double и равен 3.5. А в выражении 7/2 оба операнда принадлежат к типу int, и результат получается того же типа — он равен 3. Но даже если результат в любом случае получается целочисленным, разница все равно остается. Так, в выражении 6.0/2 операнд 6.0 имеет тип double, соответственно результат также имеет тип double и равен 3.0, но это приблизительное значение. В выражении 6/2 оба операнда имеют тип int, и результат получается того же типа — он равен 3, причем это значение точное.

Типы данных аргументов больше всего отражаются именно на выполнении оператора деления. Если один или оба операнда имеют тип double, оператор деления ведет себя, как при выполнении обычного математического деления. А вот при использовании с целочисленными операндами он просто отбрасывает дробную часть результата, возвращая только целую часть. Поэтому 10/3 равно 3 (а не 3.3333...), 5/2 — 2 (а не 2.5), а 11/3 — 3 (а не 3.6666...), то есть число не округляется — его дробная часть просто отбрасывается, сколь бы велика она ни была.

Для получения информации, потерянной при делении чисел типа int с помощью оператора /, можно воспользоваться оператором %, возвращающим остаток от целочисленного деления. Например, если разделить 17 на 5 получается 3 и 2 в остатке. Оператор / с операндами 17 и 5 возвращает 3, а оператор % возвращает 2. Поэтому код

```
cout << "17 divided by 5 is " << (17/5) << endl;
cout << "with the reminder of " << (17%5) << endl;
```

выводит такие данные:

```
17 divided by 5 is 3
with the reminder of 2
```

На рис. 2.1 показано, как работают операторы / и % для значений типа int.

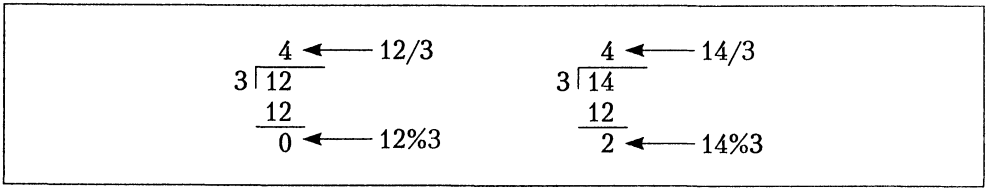


Рис. 2.1. Целочисленное деление

При использовании отрицательных значений типа int результаты выполнения операторов / и % в разных реализациях языка C++ получаются различными. Так что данные операторы следует применять только в том случае, если точно известно, что операнды не отрицательны.

Арифметические выражения могут содержать пробелы. Их можно вставлять до и после знаков операций и скобок или же не использовать вовсе. Главное, чтобы выражение получилось читабельным.

Для задания порядка выполнения операций в сложных выражениях используются скобки, как в следующих двух примерах:

```
(x + y)* z
x +(y * z)
```

При вычислении первого выражения компьютер сначала складывает x и y и затем умножает результат на z. Для вычисления второго выражения он умножает y на z и потом прибавляет к результату x. В отличие от математических выражений, в которых могут использоваться и другие виды скобок (квадратные, фигурные и т. п.), в арифметических выражениях C++ разрешается применять только круглые скобки. Другие виды скобок употребляются для иных целей.

Если выражение не содержит скобок, компьютер вычисляет его согласно *приоритету выполнения операторов*. Подобно алгебраическим правилам определяется порядок выполнения операторов * и +. Так, при вычислении выражения

```
x + y * z
```

сначала выполняется умножение, а затем сложение. За исключением некоторых стандартных случаев, таких как цепочка сложений или простое умножение внутри сложения, обычно лучше аккуратно расставлять скобки, даже если и без них выражение будет вычислено правильно. Скобки не только облегчают чтение программного кода, но и уменьшают вероятность ошибок программиста. Полный список приоритетов выполнения операторов дан в приложении 2.

Несколько примеров типичных арифметических выражений и их представления в C++ приведены в табл. 2.2.

Таблица 2.2. Арифметические выражения

Математическая формула	Выражение C++
$b^2 - 4ac$	`b*b - 4*a*c`
$x(y + z)$	`x*(y + z)`
$\frac{1}{x^2 + x + 3}$	`1/(x*x + x + 3)`
$\frac{a + b}{c - d}$	`(a + b)/(c - d)`

Ловушка: целые числа и деление

При использовании оператора / с двумя целочисленными операндами получается целое число. Это может стать причиной ошибки, если вы ожидаете получить

дробное значение. Более того, эту ошибку легко не заметить, и тогда программа, которая на первый взгляд работает прекрасно, будет выдавать неверные результаты. Предположим, что вы — ландшафтный дизайнер, проектирующий озеленение автострады. Работа оплачивается по $5000 за милю. Если вам известна длина автострады в футах, полную стоимость работ легко вычислить с помощью следующего оператора C++:

```
total_price = 5000 * (feet/5280.0);
```

Значение 5280 — количество футов в одной миле. При длине автострады 15 000 футов общая стоимость ее озеленения согласно этой формуле равна:

```
5000 * (15000/5280.0)
```

Сначала программа вычислит 15000/5280.0, и результат будет равен 2.84. Затем она умножит 5000 на 2.84 и получит 14200.00. Итак, стоимость проекта, подсчитанная нашей программой, равна $14 200.

Теперь предположим, что переменная feet имеет тип int и вы забыли ввести десятичную точку и нуль в делителе, так что оператор присваивания получился таким:

```
total_price = 5000 * (feet/5280);
```

Все по-прежнему выглядит нормально, но на самом деле у нас серьезная проблема. Если использовать эту вторую версию оператора присваивания, программа будет делить два значения типа int, и в результате feet/5280, то есть 15000/5280, окажется равным 2, а не 2.84. После чего переменной total_price будет присвоено значение 5000 * 2, то есть 10000.00. Поскольку правильным результатом является 14200.00, забытая точка обойдется вам в $4200. Обратите внимание, что результат не зависит от типа переменной total_price. Если она будет иметь тип double, ошибка все равно произойдет. А вот в случае, когда переменная feet будет иметь тип double, забытая точка не принесет вреда.

Упражнения для самопроверки

15. Преобразуйте каждую из следующих математических формул в выражение C++:

$$3x;\quad 3x+y;\quad \frac{x+y}{7};\quad \frac{3x+y}{z+2}$$

16. Что будет выведено в результате работы приведенного ниже фрагмента программного кода, входящего в состав программы, где все переменные объявлены как char?

```
a = 'b';
b = 'c';
c = a;
cout << a << b << c << 'c';
```

17. Что будет выведено в результате работы фрагмента программного кода, входящего в состав программы, где переменная number объявлена как int?

```
number = (1/3) * 3;
cout << " (1/3) * 3  is equal to " << number;
```

18. Напишите законченную программу на C++, считывающую с клавиатуры два целых числа в две переменные типа int и выводящую затем частное и остаток от целочисленного деления первого числа на второе. Воспользуйтесь операторами / и %.
19. Внимательно рассмотрите фрагмент программы, преобразующей значения температуры по Цельсию в значения по Фаренгейту, и ответьте на вопросы:

```
double c = 20;
double f;
f = (9/5) * c + 32.0;
```

а) какое значение присвоено переменной f;

б) что происходит в программе и чего, по всей вероятности, хотел добиться программист;

в) как нужно переписать программный код, чтобы реализовать намерения программиста.

Еще об операторах присваивания

Язык C++ поддерживает сокращенный синтаксис оператора присваивания для выполнения простейших арифметических операций. Его общая форма такова:

переменная операция = выражение

что эквивалентно следующему традиционному оператору:

переменная = переменная операция (выражение)

Здесь *операция* — это один из арифметических операторов, таких как +, – и т. д. Приведем несколько примеров.

Сокращенный синтаксис	Традиционный синтаксис
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total -discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rush_factor;`	`time = time / rush_factor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

2.4. Простейшее управление потоком

— Если ты думаешь, что мы из воска, выкладывай тогда денежки!
За просмотр деньги платят! Иначе не пойдет!
— И наоборот! — прибавил тот, на котором было вышито «ТРА».
— Если, по-твоему, мы живые, тогда скажи что-нибудь.

Льюис Кэрролл

Программы, которые до сих пор приводились в этой книге, состоят из простой последовательности операторов, выполняющихся в том порядке, в котором они записаны. Однако для создания более сложных программ необходимы способы

изменения порядка их выполнения. Порядок выполнения операторов часто называется *потоком управления*. В этом разделе рассматриваются механизмы ветвления, позволяющие программе выбирать одно из двух альтернативных действий в зависимости от значений переменных. Кроме того, вы познакомитесь с механизмом циклического выполнения операторов, используя который сможете многократно повторять одни и те же действия.

Простой механизм ветвления

Иногда в зависимости от введенных данных программа должна выбрать один из двух вариантов дальнейших действий. В качестве примера предположим, что вы разрабатываете программу, вычисляющую денежную сумму, которая заработана за неделю сотрудником при почасовой оплате труда. Допустим, что фирма платит сверхурочные в размере полуторной ставки за каждый час, отработанный сверх 40 часов в неделю. Если сотрудник проработал 40 или более часов, его оплата составляет

```
rate*40 + 1.5*rate*(hours - 40)
```

(В переменной rate содержится значение, соответствующее оплате за час работы, а в переменной hours — количество проработанных часов.) Однако если по каким-либо причинам сотрудник работает меньше 40 часов, в этой формуле будет получено отрицательное количество сверхурочных часов (hours-40), и результат окажется неправильным. (Для того чтобы это увидеть, достаточно выполнить вычисления со значениями rate = 1 и hours = 10. В таком случае работник получит чек с отрицательной суммой.) Правильная формула для вычисления оплаты за неделю, в течение которой сотрудник проработал меньше 40 часов, такова:

```
rate*hours
```

Если возможны оба варианта, то есть сотрудник может проработать и больше и меньше 40 часов, программе необходимо будет выбирать между двумя формулами. Иными словами, она должна выполнить такие действия:

```
Решить, верно ли условие (hours > 40)
Если да, выполнить следующий оператор присваивания:
     gross_pay = rate*40 + 1.5*rate*(hours – 40);
Если нет, выполнить такой оператор:
     gross_pay = rate*hours;
```

В C++ существует оператор, выполняющий именно такой выбор. Он называется *оператором ветвления* или оператором if...else. Например, описанное выше вычисление оплаты выполняется так:

```
if (hours > 40)
   gross_pay = rate*40 + 1.5*rate*(hours - 40);
else
   gross_pay = rate*hours;
```

Программу, использующую этот оператор, вы найдете в листинге 2.3.

На рис. 2.2 показаны две формы оператора if...else. Первая — это простая форма, применяющаяся в приведенном выше примере. Вторая описана далее в разделе

«Составные операторы». В первой форме оба оператора могут быть любыми исполняемыми операторами. Условие *логическое_выражение* (в приведенном выше примере это `hours > 40`) может быть истинно или ложно.

Когда программа достигает оператора `if...else`, выполняется один из двух входящих в его состав операторов. Если *логическое_выражение* истинно, выполняется *оператор_ДА*, в противном случае — *оператор_НЕТ*. Обратите внимание, что *логическое_выражение* должно быть заключено в скобки, этого требуют правила синтаксиса оператора `if...else`. Кроме того, заметьте, что в состав оператора `if...else` входят два простых оператора.

По одному оператору для каждой альтернативы

```
if (логическое_выражение)
   оператор_ДА
else
   оператор_НЕТ
```

Группа операторов для каждой альтернативы

```
if (логическое_выражение)
{
оператор_ДА_1
   оператор_ДА_2
      ...
   оператор_ДА_последний
}
else
{
оператор_НЕТ_1
   оператор_НЕТ_2
      ...
   оператор_НЕТ_последний
}
```

Рис. 2.2. Синтаксис оператора if...else

Листинг 2.3. Оператор if...else

```
#include <iostream>
using namespace std;
int main()
{
    int hours;
    double gross_pay, rate;

    cout << "Enter the hourly rate of pay: $";
    cin >> rate;
    cout << "Enter the number of hours worked,\n"
         << "rounded to a whole number of hours: ";
    cin >> hours;

if (hours > 40)
       gross_pay = rate*40 + 1.5*rate*(hours - 40);
    else
       gross_pay = rate*hours;
```

продолжение

Листинг 2.3 (*продолжение*)

```
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Hours = " << hours << endl;
    cout << "Hourly pay rate = $" << rate << endl;
    cout << "Gross pay = $" << gross_pay << endl;

    return 0;
}
```

Пример диалога 1

```
Enter the hourly rate of pay: $20.00
Enter the number of hours worked,
rounded to a whole number of hours: 30
Hours = 30
Hourly pay rate = $20.00
Gross pay = $600.0
```

Пример диалога 2

```
Enter the hourly rate of pay: $10.00
Enter the number of hours worked,
rounded to a whole number of hours: 41
Hours = 41
Hourly pay rate = $10.00
Gross pay = $415.00
```

Оператор `if...else` всегда содержит логическое выражение. *Логическое выражение* (называемое также *булевым выражением* или просто *условием*) — это выражение, возвращающее либо значение `true` (истина), либо значение `false` (ложь). Его простейшая форма состоит из двух чисел или переменных, сравниваемых с помощью одного из операторов сравнения, перечисленных в табл. 2.3. Обратите внимание, что некоторые операторы сравнения представлены парой символов, например ==, !=, <=, >=. Так, операция «равно» обозначается двумя знаками равенства, а операция «не равно» — символами !=. (В таких двухсимвольных операторах между символами не должен содержаться пробел.) Подряд идущие символы != воспринимаются компилятором как проверка на неравенство, с помощью данной операции при выполнении оператора `if...else` вычисляются и сравниваются два выражения. Если результатом сравнения оказывается `true`, выполняется первый оператор, а если `false` — второй.

С помощью оператора «И», представленного в C++ символами &&, можно объединить два условия. Например, следующее логическое выражение истинно, если `x` больше 2 и меньше 7:

```
(2 < x) && (x < 7)
```

Если два условия соединяются посредством оператора &&, все выражение истинно, только когда оба они истинны, в противном случае все выражение считается ложным.

Таблица 2.3. Операторы сравнения

Математический символ	Русское название операции	Обозначение в C++	Пример на C++	Математический эквивалент примера
=	Равно	==	x + 7 == 2*y	$x + 7 = 2y$
≠	Не равно	!=	ans != 'n'	$ans \neq 'n'$
<	Меньше	<	count < m + 3	$count < m + 3$
≤	Меньше или равно	<=	time <= limit	$time \leq limit$
>	Больше	>	time > limit	$time > limit$
≥	Больше или равно	>=	age >= 21	$age \geq 21$

Кроме того, два условия можно объединить с помощью оператора «ИЛИ», который в C++ обозначается символами ||. Так, следующее выражение истинно, если y меньше 0 или больше 12:

```
(y < 0) || (y > 12)
```

Если два условия объединяются оператором ||, все выражение истинно, когда истинно хоть одно из этих условий, в противном случае все выражение ложно.

Напомним, что логическое выражение, которое используется в операторе if...else в качестве критерия для выбора одного из двух действий, должно быть заключено в скобки. Например, в операторе if...else, принимающем решение на основе двух сравнений, соединенных оператором &&, скобки расставляются следующим образом:

```
if ( (temperature >= 95) && (humidity >= 90) )
    ...
```

Внутренние скобки, в которые заключены условия, не обязательны, но их применение облегчает чтение кода и поэтому обычно используется программистами.

Для отрицания логического выражения, то есть замены его значения противоположным, применяется оператор !. Выражение заключается в скобки, и знак ! помещается перед ним. Запись !(x < y) означает: «x не меньше y». Поскольку логическое выражение в операторе if...else должно располагаться в скобках, выражение с отрицанием придется дополнить еще одной парой скобок, как показано ниже:

```
if (!(x < y))
  ...
```

Обычно использования оператора ! лучше избегать. Приведенный выше оператор if...else можно переписать так:

```
if (x >= y)
  ...
```

Оператор &&

Два простых условия в логическом выражении можно объединить с помощью оператора «И», обозначаемого как &&.

Синтаксис (для логического выражения)

(условие_1) && *(условие_2)*

Пример (в операторе if...else)

```
if ( (score > 0) && (score < 10) )
    cout << "score is between 0 and 10.\n";
else
    cout << " score is not between 0 and 10.\n ";
```

Если значение переменой score больше 0 и меньше 10, будет выполнен первый оператор вывода, в противном случае — второй.

Оператор ||

Два простых условия в логическом выражении можно объединить с помощью оператора «ИЛИ», обозначаемого как ||.

Синтаксис (для логического выражения)

(условие_1) || *(условие_2)*

Пример (в операторе if...else)

```
if ( (x == 1) || (x == y) )
    cout << "x is 1 or x equals y.\n";
else
    cout << "x is neither 1 nor equal to y.\n ";
```

Если значение переменой x равно 1 или значению переменной y (либо верны оба утверждения), будет выполнен первый оператор вывода, в противном случае — второй.

До сих пор в наших примерах не было необходимости использовать оператор !, но далее в этой книге она появится, и тогда мы расскажем о нем подробнее.

Иногда логика программы требует, чтобы в случае невыполнения заданного в операторе if...else условия не выполнялись никакие действия. Эта логика реализуется с помощью упрощенной формы оператора if...else, которую часто называют просто оператором if. Вот пример:

```
if (salary >= minimum)
   salary = salary + bonus;
cout << "salary = $" << salary;
```

Первый из этих операторов — оператор if. Если значение переменной salary больше значения переменной minimum или равно ему, выполняется оператор присваивания. Если же значение salary меньше значения minimum, оператор присваивания не выполняется. На этом выполнение оператора if заканчивается и дальше выполнение программы продолжается с оператора, начинающегося словом cout, который выводит значение переменной salary уже независимо от условия оператора if.

Ловушка: сравнение нескольких значений

При необходимости сравнить в программе несколько значений ни в коем случае не используйте запись, подобную этой:

```
if (x < z < y) // НЕДОПУСТИМО
   cout << "z is between x and y.";
```

Если включить в программу такой оператор, она, скорее всего, откомпилируется и запустится, но выдаст совершенно неверные результаты. Мы расскажем, почему это происходит, когда вы немного больше узнаете о языке C++. Та же проблема возникнет и при аналогичном использовании других операторов сравнения, например, > или ==. Правильный способ записи утверждения «z больше x и меньше y» на языке C++ следующий:

```
if ( (x < z) && (z < y) ) // ПРАВИЛЬНО
   cout << "z is between x and y.";
```

Ловушка: использование = вместо ==

К сожалению, синтаксис языка C++ таков, что некоторые с виду правильные операторы делают совсем не то, чего ожидает составивший их начинающий программист. Причем с синтаксической точки зрения эти операторы верны, содержащая их программа нормально компилируется и выполняется без каких-либо сообщений об ошибках, однако результаты с точки зрения постановки задачи она выдает неверные. Эти логические ошибки находить труднее всего.

Одной из самых распространенных ошибок такого рода является использование символа = вместо символов == при сравнении двух значений. В качестве примера рассмотрим следующий оператор `if...else`:

```
if (x = 12)
   одно_действие
else
   другое_действие
```

Предположим, что вы писали его с целью проверить, содержит ли переменная `x` значение 12, и случайно ввели символ = вместо ==. Однако если вы думаете, что, проанализировав этот код, компилятор выдаст сообщение об ошибке, то ошибаетесь. Вы, скорее всего, полагаете, что операция присваивания `x = 12` не может рассматриваться как логическое выражение, поскольку она вообще не возвращает результата. Но в языке C++ присваивание является выражением, возвращающим значение, точно так же, как `x + 12` или `2 + 3`. Оно возвращает значение, присвоенное переменной слева от знака равенства. А как вы уже знаете из раздела, посвященного типу данных `bool`, значения типа `int` совместимы со значениями типа `bool` и могут быть преобразованы в значение `true` или `false`. Поскольку в нашем примере результатом присваивания является значение 12, которое не равно нулю, оно преобразуется в `true`. Таким образом, в операторе `if...else` присваивание `x = 12` рассматривается как логическое выражение, которое всегда равно `true`, так что независимо от исходного значения переменной `x` всегда выполняется первая ветвь оператора `if...else`, то есть *одно_действие*.

Эту ошибку трудно заметить, поскольку все выглядит правильным! А вот если поменять местами x и 12, так чтобы получилось (12 = x), компилятор тут же обнаружит ошибку и выдаст соответствующее сообщение. И вы добавите второй знак равенства, чтобы программа делала то, что нужно:

```
if (12 == x)
    одно_действие
else
    другое_действие
```

Итак, запомните, что использование = вместо == является очень распространенной ошибкой, не обнаруживаемой большинством компиляторов, и если такая ошибка допущена, ее трудно обнаружить. В языке C++ многие исполняемые операторы могут применяться практически в любых выражениях, включая и логические выражения в операторах if...else. Когда оператор присваивания помещается там, где ожидается логическое выражение, он интерпретируется как логическое выражение. И если только вы не сделали это намеренно, итог проверки может не соответствовать вашим ожиданиям. На первый взгляд оператор if...else будет выглядеть совершенно нормально, он откомпилируется и выполнится, но результат будет логически неправильным.

Составные операторы

Как вы уже знаете, оператор if...else позволяет задать условие, на основе которого выбирается одно из двух альтернативных действий. Каждое из них может быть представлено одним исполняемым оператором или последовательностью из нескольких таких операторов. Во втором случае (второй вариант синтаксиса, приведенного на рис. 2.2) данная последовательность операторов должна быть заключена в фигурные скобки, как в примере из листинга 2.4. Последовательность операторов, заключенная в фигурные скобки, называется *составным оператором*. Этот оператор в языке C++ интерпретируется как один оператор и может использоваться везде, где допускается применение одного оператора. Таким образом, второй вариант синтаксиса оператора if...else на рис. 2.2 является просто особым случаем первого варианта. В листинге 2.4 оператор if...else включает два составных оператора, по одному для каждой альтернативы.

Листинг 2.4. Оператор if...else с составными операторами

```
if (my_score > your_score)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish theese were golf scores\n";
    wager = 0;
}
```

Согласно синтаксису оператора if...else *оператор_ДА*, как и *оператор_НЕТ*, должен быть одним оператором. Поэтому, когда ветвление предполагает выполнение для

какой-либо из альтернатив более одного оператора, они должны быть заключены в фигурные скобки и тем самым превращены в один составной оператор. Если компилятор обнаружит между ключевыми словами if и else более одного оператора, он выдаст сообщение об ошибке. Что касается группы операторов после ключевого слова else, то к оператору if...else будет отнесен только первый из них, а остальные рассматриваются как самостоятельные операторы программы, следующие за оператором if...else.

Упражнения для самопроверки

20. Запишите оператор if...else, выводящий слово High, если значение переменной score больше 100, и слово Low в противном случае. Переменная score имеет тип int.
21. Предположим, что savings и expenses — инициализированные переменные типа double. Запишите оператор if...else, выводящий слово Solvent, уменьшающий значение переменной savings на значение переменной expenses и присваивающий переменой expenses значение 0, если значение savings не меньше значения expenses. Если же значение savings меньше expenses, этот оператор должен просто выводить слово Bankrupt, не меняя значений переменных.
22. Запишите оператор if...else, выводящий слово Passed, если значение переменной exam больше или равно 60 и значение переменной programs_done больше или равно 10. В противном случае этот оператор должен выводить слово Failed. Переменные exam и programs_done имеют тип int.
23. Запишите оператор if...else, выводящий слово Warning, если значение переменной temperature больше или равно 100 либо значение переменной pressure больше или равно 200. В противном случае этот оператор должен выводить слово OK. Переменные temperature и pressure имеют тип int.
24. Рассмотрим квадратный многочлен:

    $x^2 - x - 2$

    Условие, при котором он положителен (то есть больше нуля), определяет множество чисел, либо меньших меньшего корня (−1), либо больших большего корня (+2). Запишите логическое выражение на C++, представляющее это условие.
25. Рассмотрим квадратный многочлен:

    $x^2 - 4x + 3$

    Условие, при котором он отрицателен, определяет множество чисел, больших меньшего корня (+1) и меньших большего корня (+3). Запишите логическое выражение на C++, представляющее это условие.
26. Что выводят приведенные ниже фрагменты кода, если они выполняются в составе программы? Объясните свои ответы.

    а)
    ```
    if(0)
        cout << "0 is true";
    else
    ```

```
        cout << "0 is false";
    cout endl;
```

б)
```
if(1)
        cout << "1 is true";
    else
        cout << "1 is false";
    cout endl;
```

в)
```
if(-1)
        cout << "-1 is true";
    else
        cout << "-1 is false";
    cout endl;
```

Простые механизмы циклического выполнения

Большинство программ включает некоторые действия, повторяемые многократно. Например, программа из листинга 2.3 вычисляет оплату одного работника. Если в штате компании 100 сотрудников, программа, распечатывающая платежную ведомость, должна выполнить эти вычисления 100 раз. Часть программы, повторяющая оператор или группу операторов, называется *циклом*. В языке C++ существует несколько способов организации циклов. Одна из используемых для этого конструкций именуется оператором `while` или циклом `while`. Сначала мы продемонстрируем ее использование на коротком примере, а затем рассмотрим более сложную ситуацию.

В листинге программы 2.5 приведен простой цикл `while`, выделенный полужирным шрифтом. Часть кода, находящаяся между фигурными скобками { и } после оператора `while`, называется *телом цикла* – это повторяющиеся в цикле действия. Операторы в фигурных скобках выполняются по порядку, а затем повторяются с начала до тех пор, пока не окончится цикл. Условие окончания цикла задано в скобках после ключевого слова `while`. В первом примере диалога к программе из листинга 2.5 тело цикла выполняется трижды, и три раза выводится на экран слово `Hello`. Каждое повторение тела цикла называется *итерацией*. Таким образом, в первом примере выполняется три итерации цикла.

Листинг 2.5. Цикл while

```
#include <iostream>
using namespace std;
int main()
{
    int count_down;

    cout << "How many greetings do you want? ";
    cin >> count_down;

while (count_down > 0)
    {
        cout << "Hello ";
        count_down = count_down - 1;
    }
    cout << endl;
```

```
    cout << "That's all!\n";

    return 0;
}
```

Пример диалога 1

```
How many greetings do you want? 3
Hello Hello Hello
That's all!
```

Пример диалога 2

```
How many greetings do you want? 1
Hello
That's all!
```

Пример диалога 3

```
How many greetings do you want? 0
That's all!
```

Ключевое слово `while`, определяющее цикл, переводится с английского языка как «пока». Цикл повторяется, пока истинно логическое выражение, заданное в скобках после данного слова. В примере (см. листинг 2.5) это означает, что выполнение тела цикла повторяется, пока значение переменной `count_down` больше нуля. Давайте рассмотрим первый пример диалога и разберемся, как работает цикл `while`. Пользователь вводит значение 3, и оператор ввода присваивает его переменной `count_down`. Поэтому, когда программа достигает оператора `while`, условие выполнения цикла, согласно которому переменная `count_down` должна быть больше нуля, будет истинно. А раз так, начинается выполнение тела цикла. На каждой итерации выполняются следующие два оператора:

```
cout << "Hello ";
count_down = count_down - 1;
```

Это означает, что на экран выводится слово `Hello`, а значение переменной `count_down` уменьшается на единицу. После трех повторений значение этой переменой достигает нуля, и логическое выражение в скобках оказывается ложным. На этом цикл `while` завершается, а управление передается следующим за ним операторам, которые выводят строку `That's all!`. В результате на экране трижды повторяется слово `Hello`.

Синтаксис цикла `while` приведен на рис. 2.3. Управляющее им *логическое_выражение* ничем не отличается от выражения в операторе `if...else`. И так же как в этом операторе, оно должно быть заключено в скобки. На рис. 2.3 приведен синтаксис цикла для двух случаев: в первом из них тело цикла включает более одного оператора, а во втором — только один. Обратите внимание, что во втором случае оператор не нужно заключать в фигурные скобки.

Итак, общая идея цикла `while` понятна, теперь рассмотрим его работу подробнее. Выполнение этого цикла начинается с проверки логического выражения, следующего за ключевым словом `while`. Как и всякое другое логическое выражение, оно возвращает одно из значений: `true` либо `false`. Условие `count_down > 0` из программы,

приведенной в листинге 2.5, возвращает true, когда переменная count_down содержит положительное значение. Если выражение оказывается равным false, цикл завершается, а программа переходит к оператору, следующему за оператором while. Если же выражение истинно, выполняется тело цикла. Проверяемое выражение обычно содержит элемент, значение которого изменяется в теле цикла, такой как переменная count_down в цикле из листинга 2.5. После выполнения тела цикла логическое выражение проверяется снова. Этот процесс повторяется до тех пор, пока выражение остается истинным. Но когда после очередной итерации оно оказывается ложным, цикл while завершается.

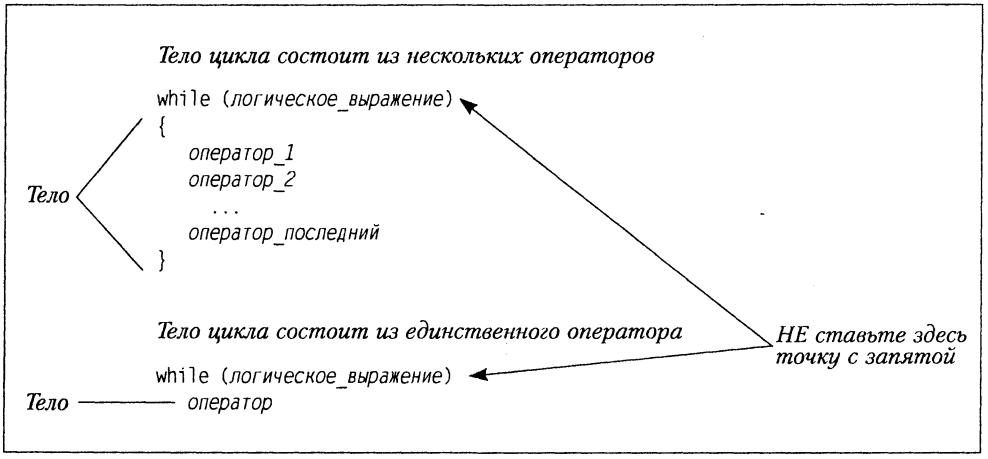


Рис. 2.3. Синтаксис оператора while

Повторим еще раз: сначала в цикле while проверяется логическое выражение. Если оно окажется ложным, тело цикла не будет выполнено ни разу. Именно эту ситуацию демонстрирует третий пример диалога к программе из листинга 2.5. Случаи, когда тело цикла при определенных условиях не должно выполняться ни разу, в программировании встречаются довольно часто. Примером может служить ситуация, когда в цикле while считывается список студентов, провалившихся на экзамене, а при этом все студенты сдали экзамен успешно.

Итак, тело цикла while может выполняться нуль раз, это бывает необходимым при определенных условиях. С другой стороны, если заранее известно, что при любых обстоятельствах тело цикла должно быть выполнено как минимум один раз, можно воспользоваться другим циклом, называемым do...while. Данный оператор подобен оператору while с той разницей, что тело определяемого им цикла всегда выполняется хотя бы один раз. Синтаксис оператора do...while приведен на рис. 2.4, а пример программы с его использованием – в листинге 2.6. Выполнение цикла do...while начинается не с проверки условия цикла, а сразу с выполнения его тела. После первой итерации он выполняется так же, как цикл while: проверяется логическое выражение, представляющее условие продолжения цикла, и если оно оказывается истинным, тело цикла выполняется снова и т. д.

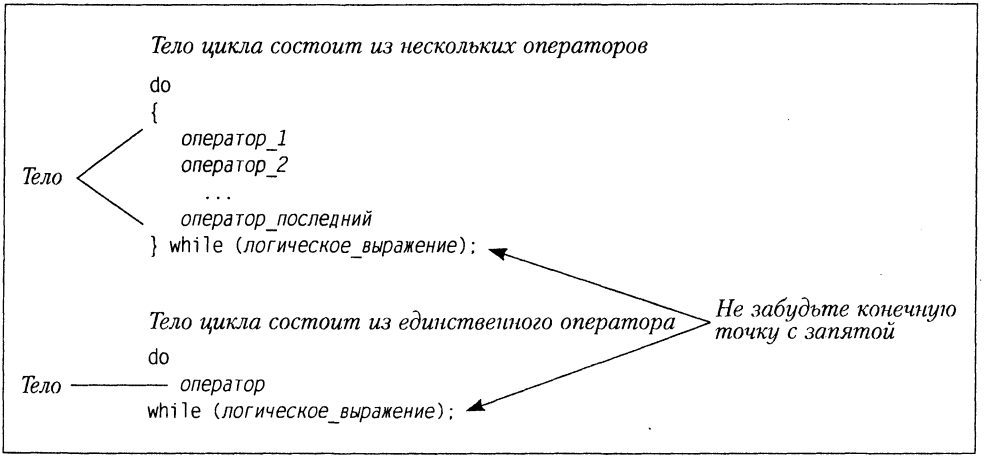


Рис. 2.4. Синтаксис оператора do...while

Листинг 2.6. Цикл do...while

```
#include <iostream>
using namespace std;
int main()
{
    char ans;

do
    {
        cout << "Hello\n";
        cout << "Do you want another greeting?\n"
             << "Press y for yes, n for no,\n"
             << "and then press Enter: ";
        cin >> ans;
    } while (ans == 'y' || ans == 'Y');

    cout << "Good-Bye\n";

    return 0;
}
```

Пример диалога

```
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press Enter: y
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press Enter: Y
Hello
Do you want another greeting?
Press y for yes, n for no,
and then press Enter: n
Good-Bye
```

Операторы инкрементирования и декрементирования

В разделе «Арифметические операторы и выражения» были описаны бинарные операторы. *Бинарным* именуются оператор, имеющий два операнда. Существуют и *унарные* операторы, имеющие по одному операнду. Два из них, + и -, вам уже знакомы; они используются в таких выражениях как +7 и -7. В языке C++ есть еще два очень популярных унарных оператора, ++ и --. Первый их них называется *оператором инкрементирования*, а второй — *оператором декрементирования*. Как правило, они используются с переменными типа `int`. Если `n` — переменная типа `int`, то оператор `n++` увеличивает ее значение на единицу, а оператор `n--` уменьшает на единицу. Операторы `n++` и `n--` являются исполняемыми. Так, операторы:

```
int n = 1, m = 7;
n++;
cout << "The value of n is changed to " << n << endl;
m--;
cout << "The value of m is changed to " << m << endl;
```

генерируют следующие выходные данные:

```
The value of n is changed to 2
The value of m is changed to 6
```

Теперь вам понятно, почему в названии языка C++ используются символы «++».

Операторы инкрементирования и декрементирования часто применяются в циклах. Например, в цикле из листинга 2.5 значение переменной `count_down` уменьшалось с помощью такого оператора:

```
count_down = count_down - 1;
```

Однако опытный программист C++ использует в этой ситуации вместо оператора присваивания оператор декрементирования, и весь цикл `while` будет выглядеть таким образом:

```
while (count_down > 0)
{
    cout << "Hello ";
    count_down--;
}
```

Пример: баланс кредитной карточки

Предположим, у вас имеется банковская кредитная карточка с балансом в $50 и банк начисляет на нее 2 % в месяц. Сколько месяцев пройдет, пока баланс на карточке превысит $100 (при условии, что вы не будете производить с ее помощью никаких оплат)? Для решения этой задачи можно написать небольшую программу.

Через месяц баланс на карточке будет равен $50 плюс 2 % от $50, то есть $51. Через два месяца он будет равен $51 плюс 2 % от $51, то есть $52,02. Через три месяца — $52,02 плюс 2 % от $52,02 и т. д. Программа может хранить текущий баланс

в переменной с именем balance. Баланс очередного месяца на основе баланса предыдущего месяца вычисляется так:

```
balance = balance + 0.02 * balance;
```

Если повторять эту операцию до тех пор, пока значение переменой balance не достигнет 100.00, а затем подсчитать количество повторений, мы узнаем, через сколько месяцев баланс составит $100. Для этого потребуется еще одна переменная, в которой будет подсчитываться количество операций увеличения баланса. Назовем ее count. Если вычисления будут производиться в цикле while, его тело будет таким:

```
balance = balance + 0.02 * balance;
count++;
```

Перед началом цикла переменным balance и count нужно присвоить соответствующие значения. Это можно сделать при их объявлении. Полная программа приведена в листинге 2.7.

Листинг 2.7. Программа, вычисляющая баланс кредитной карточки

```
#include <iostream>
using namespace std;
int main()
{
    double balance = 50.00;
    int count = 0;

    cout << "This program tells you how long it takes\n"
         << "to accumulate a debt of $100, starting with\n"
         << "an initial balance of $50 owed.\n"
         << "The interest rate is 2% per month.\n";

while (balance < 100.00)
    {
        balance = balance + 0.02 * balance;
        count++;
    }

    cout << "After " << count << " months,\n";
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "your balance due will be $" << balance << endl;

    return 0;
}
```

Пример диалога

```
This program tells you how long it takes
to accumulate a debt of $100, starting with
an initial balance of $50 owed.
The interest rate is 2% per month.
After 36 months,
your balance due will be $101.99
```

Ловушка: бесконечные циклы

Циклы `while` и `do...while` не завершаются до тех пор, пока истинно логическое выражение, заданное после слова `while`. Обычно это выражение содержит переменную, значение которой изменятся в теле цикла таким образом, что в какой-то момент выражение оказывается ложным и цикл завершается. Но если по ошибке написать программу так, что логическое выражение всегда будет оставаться истинным, цикл никогда не завершится. Такой цикл называется *бесконечным*, а состояние программы, выполняющей бесконечный цикл, именуется *зацикливанием*.

В качестве примера сначала рассмотрим нормальный цикл, завершающийся после определенного количества итераций. Следующий код выводит положительные четные числа, меньшие 12. В цикле по очереди выводятся на экран числа 2, 4, 6, 8 и 10, каждое с новой строки, после чего цикл завершается.

```
x = 2;
while (x != 12)
{
    cout << x << endl;
    x = x + 2;
}
```

Значение переменой `x` на каждой итерации цикла увеличивается на 2, пока не достигает 12. Тогда логическое выражение, заданное после слова `while`, оказывается ложным, и цикл завершается.

Теперь предположим, что мы захотели вместо четных вывести нечетные числа, меньшие 12. Решив, что для этого достаточно изменить первую строку приведенного выше кода, мы написали так:

```
x = 1;
```

Однако эта запись является ошибочной, поскольку в результате получился бесконечный цикл. После значения 11 переменной `x` присваивается значение 13, которое не равно 12, — цикл переходит эту границу и продолжается дальше, а последующие значения переменной все больше и больше отдаляются от значения 12.

Подобные проблемы типичны для циклов, в условии окончания которых выполняется проверка на равенство или неравенство (`==` или `!=`). Поэтому при работе с числами всегда безопаснее выполнять проверку на выход за определенное граничное значение. Так в нашем примере условие окончания цикла лучше всего сделать таким:

```
while (x < 12)
```

С этим условием переменную `x` можно инициализировать любым значением — цикл обязательно остановится.

Программа с бесконечным циклом работает до тех пор, пока ее не остановит какое-нибудь внешнее воздействие. Теперь, когда вы научились писать циклы и некоторые из них случайно могут получиться бесконечными, вы должны знать, как прекратить работу программы. Необходимые действия зависят от конкретной системы — во многих случаях это можно сделать, нажав комбинацию Ctrl+C (то есть, нажав и удерживая Ctrl, нажать клавишу C).

Упражнения для самопроверки

27. Что сгенерирует на выходе приведенный ниже код в программе, содержащей объявление переменой x типа int?

```
x = 10;
while (x > 0)
{
    cout << x << endl;
    x = x - 3;
}
```

28. Что будет сгенерировано на выходе в предыдущем упражнении, если вместо знака > в нем использовать знак <?

29. Что сгенерирует на выходе приведенный ниже код в программе, содержащей объявление переменой x типа int?

```
x = 10;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0);
```

30. Что сгенерирует на выходе приведенный ниже код в программе, содержащей объявление переменой x типа int?

```
x = -42;
do
{
    cout << x << endl;
    x = x - 3;
} while (x > 0);
```

31. Каково важнейшее отличие оператора while от do...while?

32. Что сгенерирует на выходе приведенный ниже код в программе, содержащей объявление переменой x типа int?

```
x = 10;
while (x > 0)
{
    cout << x << endl;
    x = x + 3;
}
```

33. Напишите законченную программу на C++, выводящую числа от 1 до 20, каждое с новой строки. Больше эта программа ничего делать не должна.

2.5. Некоторые особенности оформления программ

В важных вопросах главное не искренность, а стиль.

Оскар Уайльд

В наших демонстрационных программах имена переменных отражают их назначение, а сами программы одинаково отформатированы. Например, объявления

переменных и операторы записываются с равными отступами. Все эти и другие составляющие стиля программирования важны отнюдь не только с эстетической точки зрения. Программа, написанная в хорошем стиле, более читабельна, ее легче исправлять и модифицировать.

Отступы

Строки программы должны быть расположены так, чтобы логически связанные элементы выглядели единой группой. Для этого можно пропускать строку между частями программы, которые логически отделены друг от друга. Кроме того, сделать структуру программы более наглядной помогают отступы. Если один оператор вложен в другой, его следует сдвинуть вправо. Это, в частности, касается операторов `if...else`, `while` и `do...while`, содержащих один или несколько вложенных операторов. Примеры их расположения вы можете увидеть в листингах, приведенных в этой книге.

Очень важный элемент, определяющий структуру программы, — фигурные скобки. Если каждая скобка размещается на отдельной строке и при этом соответствующие друг другу скобки расположены в точности друг под другом, их легко находить визуально. Обратите внимание, что в наших примерах перед некоторыми парами скобок имеются отступы. Они показывают, что одна пара скобок вложена в другую. Так, в листинге 2.7 скобки, в которые заключено тело цикла `while`, сдвинуты вправо по отношению к скобкам, охватывающим главную часть программы.

Существует как минимум два распространенных мнения относительно того, где следует размещать фигурные скобки. Часть программистов, включая и автора этой книги, считает, что их лучше располагать в отдельных строках, поскольку это делает программу более читабельной. Другие программисты полагают, что открывающую скобку не нужно выносить в отдельную строку. При известной аккуратности второй метод помогает сделать программу более компактной. Вы, конечно, можете поступать по своему усмотрению, и выбрать стиль, который кажется вам наиболее удобным, но если вам придется дорабатывать чью-либо программу, придерживайтесь стиля ее автора, чтобы весь исходный код программы был единообразным. Кроме того, учтите, что резкие отличия вашего стиля программирования от общепринятых норм приведут к трудностям при чтении написанных вами программ другими людьми.

Комментарии

Для того чтобы сделать программу более понятной, нужно сопроводить ее важнейшие фрагменты некоторыми пояснениями. Такие пояснения называются *комментариями*. C++, как и большинство других языков программирования, предоставляет синтаксис для включения комментариев в программы. Символы // (две косые черты без пробела между ними) в C++ отмечают начало комментария. Весь текст от символа // и до конца строки считается комментарием и игнорируется компилятором. Для размещения в программе комментария длиной в несколько строк нужно просто в начало каждой строки поместить названные символы.

Чтобы комментарии отличались от текста исполняемого кода, многие редакторы выделяют текст комментария другим цветом.

Язык C++ поддерживает еще один альтернативный синтаксис комментариев, когда комментарием считается любой текст между символами /* и */. Этот синтаксис особенно удобен для многострочных комментариев, поскольку в отличие от синтаксиса с использованием двух косых не требует, чтобы помечалась каждая строка комментария:

```
/* Это комментарий, занимающий три строки.
   Обратите внимание, что во второй строке
   нет никаких символов комментария.*/
```

Комментарии такого типа можно вставлять в любое место программы, где допускается пробел или разрыв строки. Но размещать их следует не как попало, а так, чтобы программа оставалась читабельной и не нарушалась ее визуальная структура. Обычно комментарии располагают в конце строки или на отдельных строках.

О том, какой из двух типов комментариев (// или /* */) более удобен, существуют разные мнения. При аккуратном использовании удобны оба типа.

Трудно сказать, сколько комментариев должна содержать программа. Единственно правильный ответ — необходимое и достаточное количество, но он мало что говорит программисту-новичку. Чувство меры в отношении комментариев индивидуально и приходит с опытом. Если что-либо важно, но не очевидно, требуется комментарий. Однако их не должно быть ни слишком много ни слишком мало, поскольку и то и другое затрудняет чтение программы. Программа с малым количеством комментариев может быть непонятна, а с комментариями в каждой строчке настолько перегружена, что ее структура теряется в массе очевидных сентенций. Например, комментарии, подобные следующему, ничего не добавляют к имеющейся информации:

```
distance = speed * time // Вычисляет пройденное расстояние.
```

Обратите внимание на комментарий в начале листинга 2.8 — с аналогичных комментариев должна начинаться любая программа. Здесь приведена вся важнейшая информация: в каком файле хранится программа, кто ее автор, как с ним связаться, что делает программа, когда она в последний раз модифицировалась и прочие полезные сведения. Набор сведений, сообщаемых в заголовочном комментарии, зависит от конкретной ситуации. Остальные программы, приведенные в этой книге, не будут содержать таких длинных комментариев, однако вам обязательно следует взять за правило начинать каждую программу с подобного комментария со стандартным содержанием.

Листинг 2.8. Комментарии и именованные константы

```
// Программа должна всегда начинаться с подобного комментария.
// Имя файла: health.cpp (В вашей системе может требоваться
// суффикс, отличный от cpp.)
// Автор: здесь пишется ваше имя.
// Адрес электронной почты: you@yourmachine.bla.bla
// Номер задания: 2
// Описание: программа, определяющая, здоров ли пользователь.
// Последнее изменение: 23 сентября 2004 г.
#include <iostream>
using namespace std;
int main()
```

продолжение

Листинг 2.8 (*продолжение*)

```
{
const double NORMAL = 98.6;   // Градусов по Фаренгейту
double temperature;

    cout << "Enter your temperature: ";
    cin >> temperature;

    if (temperature > NORMAL)
    {
        cout << "You have a fever.\n";
        cout << "Drink lots of liquids and get to bed.\n";
    }
    else
    {
        cout << "You don't have a fever.\n";
        cout << "Go study.\n";
    }

    return 0;
}
```

Пример диалога

```
Enter your temperature: 98.6
You don't have a fever.
Go study.
```

Именованные константы

С применением чисел в компьютерных программах связаны две проблемы. Первая заключается в том, что числа не имеют семантического значения. Например, когда в программе встречается число 10, вы можете ничего не знать о его смысле — в банковской программе это может быть количество офисов или окон в главном офисе. Чтобы понять программу, нужно знать значения всех используемых в ней констант. Вторая проблема состоит в следующем: применяемые в программе числа могут изменяться, и тогда приходится вносить в нее различные модификации, что всегда сопряжено с риском возникновения ошибок. Предположим, что в банковской программе число 10 встречается двенадцать раз. При этом четыре раза оно применяется для обозначения количества офисов, и восемь раз — количества окон в главном офисе. Как только банк открывает новое подразделение, программу приходится обновлять. И тогда может возникнуть ситуация, когда одно из чисел, которое должно было быть заменено числом 11, не будет изменено или, наоборот, будет изменено число, которое не нужно было трогать. Во избежание таких ошибок можно присвоить каждому из чисел имя. В банковской программе можно ввести константы с именами BRANCH_COUNT и WINDOW_COUNT, которые будут иметь значение 10. При открытии нового офиса достаточно будет изменить программу только в одном месте — там, где объявляется константа BRANCH_COUNT.

Как же в программе на C++ назначить имя числу? Один из способов — создать переменную и инициализировать ее этим значением, как в следующем примере:

```
int BRANCH_COUNT = 10;
int WINDOW_COUNT = 10;
```

Но у данного способа имеется недостаток: где-нибудь в программе можно случайно изменить значение переменной. Поэтому C++ предоставляет программисту возможность пометить инициализированную переменную как не подлежащую изменению. Когда программа попытается изменить одну из таких переменных, это вызовет ошибку. Чтобы пометить переменную как неизменяемую, нужно в начало ее объявления поместить ключевое слово const (сокращение от англ. constant — константа). Например:

```
const int BRANCH_COUNT = 10;
const int WINDOW_COUNT = 10;
```

Здесь переменные имеют один и тот же тип, и их объявления можно объединить в одно:

```
const int BRANCH_COUNT = 10, WINDOW_COUNT = 10;
```

Однако большинство программистов предпочитает помещать объявление каждого имени в отдельную строку. Слово const часто именуют *квалификатором*.

Переменную, объявленную с квалификатором const, обычно называют *именованной константой*. Написание идентификаторов именованных констант в верхнем регистре с точки зрения синтаксиса языка C++ не обязательно, однако по общему соглашению так поступают все программисты.

После объявления константы ее можно применять везде, где допускается использование числа, и она будет иметь в точности то же значение, что и представляемое ею число. Для изменения именованной константы достаточно изменить ее значение в объявлении. Так, если в объявлении константы BRANCH_COUNT заменить значение 10 значением 11, везде в тексте программы она будет иметь значение 11.

Хотя в программе можно использовать и неименованные числовые константы, это не рекомендуется. Разве что речь идет об общеизвестном значении, которое не может измениться, таком как 100 сантиметров в 1 метре. Но все остальные числовые константы следует определять описанным выше способом, что облегчит и чтение программы, и ее последующую модификацию.

Простая программа, где демонстрируется применение квалификатора const, приведена выше в листинге 2.8.

Именованные константы, объявленные с квалификатором const

Инициализируя переменную непосредственно в ее объявлении, можно запретить изменение ее значения. Для этого в объявление нужно добавить ключевое слово const.

Синтаксис

```
const имя_типа имя_переменной = константа;
```

Пример

```
const int MAX_TRIAE = 3;
const double PI = 3.14159;
```

Упражнения для самопроверки

34. Приведенный ниже оператор `if...else` будет благополучно откомпилирован и выполнен. Однако отступы и разрывы строк расположены в нем не так, как принято в наших программах. Перепишите его согласно принятому в этой книге стилю.

```
if (x < 0) {x = 7; cout << "x is now positive.";}else
{x = -7; cout << "x is now negative.";}
```

35. Что выведут приведенные ниже строки, включенные в состав программы?

```
// cout << "Hello from";
cout << "Self-Test Exercise";
```

36. Напишите программу, запрашивающую у пользователя количество галлонов и выводящую соответствующее количество литров (в 1 галлоне 3,78533 литра). Используйте именованную константу. Поскольку это только упражнение, комментарии в нем не требуются.

Резюме

- Присваивайте переменным информативные имена.
- Следите за тем, чтобы переменным назначались подходящие типы данных.
- Инициализируйте переменные до того, как программа попытается использовать их значения. Инициализацию переменной можно выполнить либо прямо в ее объявлении, либо с помощью оператора присваивания до ее первого применения.
- Используйте скобки в арифметических выражениях, чтобы порядок выполнения операций был более наглядным.
- Когда требуется, чтобы пользователь ввел какое-либо значение, обязательно выводите достаточно информативный текст, запрашивающий это значение, а затем дублируйте введенное значение на экране.
- Помните, что оператор `if...else` позволяет программе выбрать одно из двух альтернативных действий, а оператор `if` позволяет программе решить, следует ли выполнять заданное действие.
- Не забывайте, что тело цикла `do...while` всегда выполняется как минимум один раз, а тело цикла `while` в некоторых случаях может не выполниться ни разу.
- Присваивайте числовым константам в программе вместо чисел информативные имена. Такие константы объявляются как переменные с квалификатором `const`.
- При написании программ соблюдайте отступы, расставляйте пробелы и пустые строки, используя в качестве образца приведенные в этой книге примеры.
- Снабжайте программы комментариями, поясняющими их ключевые элементы и непонятные фрагменты.

Ответы к упражнениям для самопроверки

1.
```
int feet = 0, inches = 0;
int feet(0), inches(0);
```

2.
```
int count = 0;
double distance = 1.5;
```

   или

```
int count(0);
double distance(1.5);
```

3.
```
sum = n1 + n2;
```

4.
```
length = length + 8.3;
```

5.
```
product = product * n;
```

6. Выходные данные такой программы зависят от системы и ее предшествующего использования.

```
#include <iostream>
using namespace std;
int main()
{
    int first, second, third, fourth, fifth;
    cout << first << " " << second <<  " " << third
         << " " << fourth << " " << fifth << endl;

    return 0;
}
```

7. На этот вопрос нет единственного ответа. Вот возможные варианты:

   а) `speed`;

   б) `pay_rate`;

   в) `max_score`.

8.
```
cout << "The answer to the question of\n"
     << "Life, the Universe, and Everything is 42.\n";
```

9.
```
cout << "Enter a whole number and press Enter: ";
cin >> the_number;
```

10.
```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(3);
```

11.
```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello world\n";
    return 0;
}
```

12.
```
#include <iostream>
using namespace std;
int main()
{
    int n1, n2, sum;
    cout << "Enter two whole numbers\n";
    cin >> n1  >> n2;
    sum = n1 + n2;
    cout << "The sum of " << n1 <<  " and " << n2
         << " is " << sum << endl;

    return 0;
}
```

13.
```
cout << endl << "\t";
```

14.
```
#include <iostream>
using namespace std;
int main()
{
    double one(1.0), two(1.414), three(1.732), four(2.0),
           five(2.236);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(3);

    cout << "\tN\tSquare Root\n";
    cout << "\t1\t" << one << endl
         << "\t2\t" << two << endl
         << "\t3\t" << three << endl
         << "\t4\t" << four << endl
         << "\t5\t" << five << endl;
    return 0;
}
```

15.
```
3*x
3*x + y
(x + y)/7   // Обратите внимание, что ответ x + y/7 будет неправильным.
(3*x + y)/(z + 2)
```

16.
```
bcbc
```

17.
```
(1/3) * 3 is equal to 0
```

Поскольку значения 1 и 3 имеют тип int, оператор / выполняет целочисленное деление, при котором отбрасывается остаток, так что 1/3 равно 0, а не 0.3333... . Поэтому и все выражение будет равно 0 * 3, то есть 0.

18.
```
#include <iostream>
using namespace std;
int main()

{
    int number1, number2;

    cout << "Enter two two whole numbers: ";
    cin >> number1 >> number2;
```

```
    cout << number1 << " divided by " << number2
         << " equals " << (number1/number2) << endl
         << "with the reminder of " << (number1%number2)
         << endl;

    return 0;

}
```

19. а) `52.0;`

    б) в результате операции 9/5 получается значение 1 типа `int`. Поскольку делимое и делитель имеют тип `int`, для них выполняется целочисленное деление и дробная часть отбрасывается;

    в) `f = (9.0/5) * c + 32.0;`

    или

    `f = 1.8 * c + 32.0;`

20.
```
if (score > 100)
    cout << "High";
else
    cout << "Low";
```

В конец обеих строк можно было бы добавить `\n` в зависимости от конкретной программы.

21.
```
if (savings >= expenses)
{
    savings = savings - expenses;
    expenses = 0;
    cout << "Solvent";
}
else
{
    cout << "Bankrupt";
}
```

В конец обеих строк можно было бы добавить `\n` в зависимости от конкретной программы.

22.
```
if ( (exam >= 60) && (programs_done >= 10) )
    cout << "Passed";
else
    cout << "Failed";
```

В конец обеих строк можно было бы добавить `\n` в зависимости от конкретной программы.

23.
```
if ( (temperature >= 100) || (pressure >= 200) )
    cout << "Warning";
else
    cout << "OK";
```

В конец обеих строк можно было бы добавить `\n` в зависимости от конкретной программы.

24. `(x < -1) || (x > 2)`
25. `(1 < x) && (x < 3)`
26. а) `0 is false` (в разделе «Совместимость типов данных» рассказывалось, что значение `0` типа `int` преобразуется в `false`);

    б) `1 is true` (в разделе «Совместимость типов данных» рассказывалось, что любое ненулевое значение типа `int` преобразуется в `true`);

    в) `-1 is true` (в разделе «Совместимость типов данных» рассказывалось, что любое ненулевое значение типа `int` преобразуется в `true`).

27.
```
10
7
4
1
```

28. Эта программа ничего не выведет, поскольку логическое выражение `(x < 0)` с самого начала ложно, поэтому оператор `while` сразу завершится и тело цикла не выполнится ни разу.
29. Выходные данные будут точно такими же, как в упражнении 27.
30. Тело цикла выполняется до проверки логического выражения. Это выражение ложно, поэтому выходные данные таковы:

```
-42
```

31. В цикле `do...while` тело цикла всегда выполняется хотя бы один раз. Для цикла `while` возможны условия, при которых тело цикла не выполняется ни разу.
32. Это бесконечный цикл. Результаты его начинаются со следующих данных:

```
10
13
16
19
```

(Когда значение переменной `x` превысит максимальное целочисленное значение, допускаемое вашим компьютером, программа может остановиться или повести себя каким-либо иным неожиданным образом, но концептуально цикл остается бесконечным.)

33.
```
#include <iostream>

using namespace std;
int main()

{
    int n = 1;
    while (n <= 20)
    {
        cout << n << endl;
        n++;
    }

    return 0;
}
```

34.
```
if (x < 0)
{
```

```
    x = 7;
    cout << "x is now positive.";
}
else
{
    x = -7;
    cout << "x is now negative.";
}
```

35. Первая строка — это комментарий, она не выполняется. Поэтому данный код выведет только следующую строку:

```
Self-Test Exercise
```

36.
```
#include <iostream>
using namespace std;
int main()

{
    const double LITERS_PER_GALLON = 3.78533;
    double gallons, liters;

    cout << "Enter the number of gallons:\n";
    cin >> gallons;

    liters = gallons*LITERS_PER_GALLON;
    cout << "There are " << liters << " liters in "
         << gallons << " gallons.\n";

    return 0;
}
```

Практические задания

1. В метрической системе единиц 1 тонна равна 35 273,92 унции. Напишите программу, считывающую значение веса коробки сухих завтраков в унциях и выводящую его в тоннах, а также количество коробок, содержащих одну тонну сухих завтраков. Программа должна позволять пользователю повторять эти вычисления столько раз, сколько ему будет необходимо.
2. В результате исследований было установлено, что искусственный заменитель сахара, обычно используемый в малокалорийном лимонаде, вызвал смерть лабораторной мыши. Ваш друг очень хочет похудеть, но не может отказать себе в лимонаде. Поэтому он решил выяснить, сколько лимонада можно выпить, не опасаясь летального исхода. Напишите программу, отвечающую на этот вопрос. На входе ее задается количество заменителя сахара, смертельное для мыши, вес мыши и вес человека. Для безопасности указывается не нынешний вес человека, желающего похудеть, а тот вес, который он считает для себя нормальным. Предположим, что лимонад содержит 0,1 % сахарозаменителя. Присвойте этому числу (которое можно выразить как значение `0.001` типа `double`) имя, используя квалификатор `const`. Программа должна позволять пользователю повторять вычисления произвольное количество раз.

3. Работникам некоторой компании увеличена заработная плата, причем повышение охватывает и шесть уже прошедших месяцев. Напишите программу, которая принимает на входе значение, соответствующее окладу сотрудника за предыдущий год и выводит дополнительную сумму, которая должна быть ему выплачена за прошлый год, новый годовой оклад и новый месячный оклад. Значение увеличения оклада задайте в программе в виде именованной константы, объявленной с квалификатором `const`. Программа должна позволять пользователю повторять вычисления произвольное количество раз.

4. Механизм начисления процентов по ссуде не всегда прост. Одной из форм ссуды является ссуда с выплатой по частям, действующая следующим образом. Предположим, что номинальная сумма ссуды составляет $1000, процентная ставка равна 15 %, а срок погашения – 18 месяцам. Проценты вычисляются путем умножения $1000 на 15 %, что составляет $150. Эта цифра умножается на срок погашения ссуды, который равен 1,5 года, так что общая сумма процентов к выплате составляет $225. Данная сумма сразу вычитается из номинальной суммы ссуды, так что потребитель получает на руки только $775. Возврат ссуды производится равными частями (вычисляемыми на основе номинальной суммы) ежемесячно. Таким образом, месячная выплата равна результату деления $1000 на 18, то есть $55,56. Этот метод вычисления может быть не так уж плох, если потребителю нужно $775 долларов, но если ему требуется ровно $1000, вычисления несколько усложняются. Напишите программу, которая принимает три значения: сумму, необходимую потребителю, процентную ставку и срок погашения ссуды. Программа должна вычислить номинальное значение и сумму месячной выплаты и позволять пользователю повторять вычисления произвольное количество раз.

5. Напишите программу, определяющую, соответствует ли актовый зал правилам пожарной безопасности с точки зрения максимальной вместимости комнаты. Программа должна получать значения максимальной вместимости и количества людей, приглашенных на собрание. Если количество людей не превосходит максимальную вместимость зала, программа сообщает, что проведение собрания в этом помещении допустимо, и указывает, сколько еще человек может на него прийти. Если же количество людей превышает максимальную вместимость, программа сообщает, что согласно правилам пожарной безопасности собрание не может быть проведено и указывает, на сколько нужно сократить количество его участников. Усложненная версия программы должна позволять пользователю повторять вычисления произвольное количество раз.

6. Работник получает $16,78 в час за установленное количество рабочих часов в неделю. Сверхурочные оплачиваются в полуторном размере. Из общей заработанной суммы производятся следующие отчисления: 6 % на социальные нужды, 14 % в качестве подоходного налога в федеральный бюджет, 5 % в качестве подоходного налога в бюджет штата и $10 в неделю в качестве профсоюзного взноса. Если на содержании работника находятся три или более членов семьи, из его заработной платы удерживается еще $35 на покрытие дополнительных расходов по страхованию здоровья. Напишите программу, считывающую количество проработанных часов за неделю и количество членов

семьи работника, находящихся на его содержании, и выводящую заработанную сумму, сумму каждого из вычетов и сумму к выдаче. Более сложная версия этой программы должна позволять пользователю повторять вычисления произвольное количество раз.

7. Из-за нестабильности цен трудно составить бюджет на несколько лет. Если вашей компании требуется 200 карандашей в год, нельзя просто взять текущую цену карандаша и использовать ее для вычисления стоимости закупок на два ближайших года. Из-за инфляции уже в будущем году эта цена, по всей вероятности, будет выше нынешней. Напишите программу, вычисляющую ожидаемую стоимость товара через заданное количество лет. Программа должна запрашивать текущую цену товара, количество лет и коэффициент инфляции, а выводить ожидаемую стоимость товара. Коэффициент инфляции вводится пользователем в виде значения, выраженного в процентах, например 5,6 %. Программа должна преобразовать это число в множитель (в данном случае 0,056) и в цикле прибавить повышение цены из-за инфляции.

   Подсказка. Это похоже на вычисление процентов по кредитной карточке, как в примере, рассматривавшемся в разделе 2.4.

8. Вы только что приобрели стереосистему стоимостью $1000 на следующих условиях. Никаких начальных взносов, кредитная ставка 18 % годовых (то есть 1,5 % в месяц) и ежемесячная выплата $50. Из ежемесячных $50 погашаются проценты, а оставшаяся сумма идет на выплату части долга. Поэтому в первый месяц вы платите 1,5 % от $1000, то есть $15. Остальные $35 вычитаются из вашего долга, после чего вы остаетесь должны $965,00. Во втором месяце вы платите 1,5 % от $965,00, то есть $14,48. Долг уменьшается на $35,52 ($50 – $14,48) и т. д. Напишите программу, вычисляющую, сколько месяцев уйдет на выплату кредита, а также общую сумму процентов, которую вы выплатите за это время. Используйте цикл для вычисления суммы процентов и оставшейся суммы долга за каждый месяц. (Окончательная версия программы не должна выводить проценты, выплачиваемые каждый месяц, и сумму оставшегося долга, однако в промежуточной версии полезно вывести эти цифры, чтобы контролировать правильность вычислений.) Для подсчета количества итераций цикла и соответственно количества месяцев, в течение которых выплачивается долг, используйте переменную. Возможно, вам потребуются и другие переменные. Последняя выплата может быть меньше $50, в зависимости от суммы долга, но не забудьте включить в нее проценты. Если в конце вы останетесь должны $50, то сумма выплат за очередной месяц не покроет долга и выплата будет продолжена еще на месяц. Проценты с денежной суммы в $50 за месяц составят 75 центов.

9. Напишите программу, считывающую 10 целых чисел и выводящую сумму тех из них, которые оказались больше нуля, сумму всех остальных чисел (отрицательных или равных нулю) и полную сумму введенных чисел (положительных, отрицательных и равных нулю). Пользователь вводит эти 10 чисел только один раз в любом порядке. Программа не должна просить его ввести положительные и отрицательные числа по отдельности.

Глава 3 Процедурная абстракция и функции

Там был также весьма изобретательный архитектор, придумавший новый способ постройки домов, начиная с крыши и кончая фундаментом.

Джонатан Свифт

Программу можно условно разделить на нескольких составляющих, например, получение начальных данных, вычисления и вывод выходных данных. Как и многие другие языки, C++ позволяет программировать эти части, называемые в данном языке функциями, по отдельности и назначать им собственные имена. В главе 3 будет рассмотрен синтаксис одного из двух базовых типов функций C++, а именно: функций, предназначенных для вычисления единственного значения. Кроме того, мы расскажем о роли функций в проектировании программы.

3.1. Нисходящее проектирование

Как вы помните, работа над программой начинается с выработки метода решения поставленной задачи и представления его в виде инструкций на русском языке, предназначенных человеку, а не компьютеру. Как рассказывалось в главе 1, такой набор инструкций называется алгоритмом. Наилучший подход к разработке алгоритма следующий: разбить начальную задачу на несколько подзадач, каждую из них разбить на меньшие подзадачи и т. д. Наконец будут выделены подзадачи столь малого размера, что реализовать их на языке C++ окажется совсем просто. Данная методика называется *нисходящим проектированием* (иногда ее также именуют *пошаговой детализацией* или методикой *«разделяй и властвуй»*).

Следуя этому методу, сначала разрабатывают алгоритмы, а затем реализуют их в виде программ и подпрограмм, сохраняя структуру «задача-подзадача». В результате получается понятная программа, которую в дальнейшем легко модифицировать, не говоря уже о том, что ее гораздо проще писать, тестировать и отлаживать.

Как и большинство других языков программирования, C++ предоставляет специальные средства для реализации подзадач в виде отдельных частей программы. Такие части, в иных языках называемые подпрограммами или процедурами, в C++ именуются *функциями*.

Одно из преимуществ структурирования программы в виде набора функций — возможность распределения работы над проектом между несколькими программистами, пишущими каждый свою группу функций. Это особенно важно при создании крупных программ, которые невозможно разработать в одиночку в короткий срок, таких как компиляторы или офисные системы.

3.2. Стандартные функции

В состав C++ входят библиотеки стандартных функций, которые можно использовать в разрабатываемых программах. Поэтому прежде чем говорить о написании собственных функций, мы покажем, как пользоваться готовыми.

Использование функций

В качестве примера готовой функции воспользуемся функцией `sqrt`, вычисляющей квадратный корень из заданного числа. Эта функция берет число, например `9.0`, и вычисляет его квадратный корень, в данном случае `3.0`. Значение, передаваемое функции при вызове, называется ее *аргументом*, а то, которое она вычисляет, — *возвращаемым значением*. Функция может иметь более одного аргумента, но возвращаемое значение у нее только одно. Ее можно представить как маленькую программу, аргументы которой аналогичны входным данным, а возвращаемое значение — выходным данным.

Пользоваться функцией в программе очень просто. Например, чтобы присвоить переменной `the_root` значение квадратного корня из `9.0`, нужно выполнить следующий оператор:

```
the_root = sqrt(9.0);
```

Выражение `sqrt(9.0)` называется *вызовом функции*. Аргументом в нем может быть константа, такая как `9.0`, переменная или более сложное выражение. Вызов функции является обыкновенным выражением, его можно использовать везде, где допускается применение выражения того типа, к которому относится возвращаемое функцией значение. Например, значение, возвращаемое функцией `sqrt`, имеет тип `double`. Таким образом, вполне допустимо присваивание

```
bonus = sqrt(sales)/10;
```

Здесь `sales` и `bonus` — переменные типа `double`. Вызов функции `sqrt` определяется одним элементом; его можно заключить в скобки, но это не обязательно:

```
bonus = (sqrt(sales))/10;
```

Вызов функции может использоваться в операторе `cout`, как в следующем примере:

```
cout << "The side of a square with area "
     << area << " is " << sqrt(area);
```

Вызов функции

Вызов функции — это выражение, состоящее из имени функции, за которым следуют аргументы, заключенные в круглые скобки. Если у функции более одного аргумента, их разделяют запятыми. Вызов функции может использоваться как любое другое выражение, тип которого соответствует типу возвращаемого ею значения.

Синтаксис

имя_функции(список_аргументов)

Здесь *список_аргументов* — разделенный запятыми список аргументов

аргумент_1, аргумент_2, ..., аргумент_последний

Примеры

```
side = sqrt(area);
cout << "2.5 to the power 3.0 is "
     << pow(2.5, 3.0);
```

В листинге 3.1 приведена программа, где используется стандартная функция sqrt. Программа вычисляет максимальный размер собачьей будки, которую можно построить при наличии заданной суммы денег. У пользователя запрашивается эта сумма, а затем вычисляется площадь будки в квадратных футах. С помощью функции sqrt рассчитывается длина стены будки, имеющей квадратную форму.

Листинг 3.1. Вызов функции

```
// Вычисляет размер будки, которую можно построить
// при наличии указанного количества денег.
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    const double COST_PER_SQ_FT = 10.50;
    double budget, area, length_side;

    cout << "Enter the amount budgeted"
         << " for your dog house $";
    cin >> budget;

    area = budget/COST_PER_SQ_FT;
    length_side = sqrt(area);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For a price of $" << budget << endl
         << "I can build you a luxurious square"
         << "dog house\n"
         << "that is " << length_side
         << " feet on each side.\n";

    return 0;
}
```

Пример диалога

```
Enter the amount budgeted for your dog house $25.00
For a price of $25.00
I can build you a luxurious square dog house
that is 1.54 feet on each side.
```

Обратите внимание на новый элемент в листинге программы 3.1:

```
#include <cmath>
```

Эта строка очень похожа на уже знакомую вам

```
#include <iostream>
```

Фактически это строки одного типа. Как говорилось в главе 2, они называются директивами include. Имя в угловых скобках (<>) — имя *заголовочного файла*. Заголовочный файл библиотеки предоставляет компилятору базовую информацию об этой библиотеке, а директива include включает данный файл в программу. Все это позволяет компоновщику найти объектный код содержащихся в библиотеке функций и правильно связать библиотеку с программой. Например, библиотека iostream содержит определения операторов cin и cout; ее заголовочный файл называется iostream. Библиотека cmath включает определение функции sqrt и множества других математических функций; ее заголовочный файл именуется cmath. Если в программе используется функция из какой-нибудь библиотеки, такая программа должна содержать директиву include с именем заголовочного файла этой библиотеки, подобную следующей:

```
#include <cmath>
```

Точно следуйте синтаксису примеров, приведенных в этой книге. Не забывайте применять угловые скобки (между ними и именем файла не должны стоять пробелы). Кроме того, некоторые компиляторы требуют, чтобы возле символа # тоже не было пробелов, поэтому лучше не использовать их ни в начале строки, ни между символом # и словом include. Обычно директивы include помещают в самое начало файла, содержащего программу.

Как говорилось ранее, директива

```
#include <iostream>
```

требует наличия следующей директивы using:

```
using namespace std;
```

поскольку определения таких имен как cin и cout относятся к пространству имен std. То же касается и большинства других стандартных библиотек. Если в программе есть директива include с именем какой-нибудь стандартной библиотеки вроде cmath, нужно включить в нее и директиву using.

Независимо от количества директив include в программе необходима только одна директива using. Как правило, для использования библиотеки этих двух директив достаточно. Если после их включения в программу все работает нормально, то никаких дополнительных действий не требуется. Однако для подключения некоторых библиотек нужны дополнительные директивы компилятора или явный запуск программы-компоновщика. Ранние компиляторы C и C++ не выполняли автоматического поиска библиотек для связывания. Детали процесса в разных

системах различны, поэтому в случае возникновения проблем обратитесь к специалисту по вашей системе.

Возможно, вы слышали от кого-нибудь, что директивы include обрабатываются не компилятором, а препроцессором. Это верно, но различие между этими программными компонентами не имеет для нас особого значения, поскольку практически все компиляторы автоматически вызывают препроцессор во время компиляции программы.

Несколько стандартных библиотечных функций описано в табл. 3.1, еще некоторые — в приложении 4. Обратите внимание, что функции, возвращающие абсолютное значение заданного числа (abs и labs), находятся в библиотеке, заголовочный файл которой называется stdlib, поэтому любая использующая их программа должна содержать следующую директиву:

```
#include <stdlib>
```

Все остальные перечисленные в таблице функции (в частности, sqrt) определены в библиотеке с заголовочным файлом cmath.

Таблица 3.1. Некоторые стандартные библиотечные функции

Имя	Описание	Тип аргумента	Тип возвращаемого значения	Пример	Значение	Заголовочный файл
sqrt	Квадратный корень	double	double	sqrt(4.0)	2.0	cmath
pow	Возведение в степень	double	double	pow(2.0, 3.0)	8.0	cmath
abs	Абсолютное значение для int	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	Абсолютное значение для long	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	Абсолютное значение для double	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	Округление к большему целому числу	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Округление к меньшему целому числу	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath

Кроме того, обратите внимание на наличие трех функций, возвращающих абсолютное значение. Функция abs возвращает абсолютное значение числа типа int, функция labs — числа типа long, а функция fabs — числа типа double. При этом первые две находятся в библиотеке с заголовочным файлом stdlib, а функция fabs — в библиотеке с заголовочным файлом cmath. (Напомним, что числа с дробной

частью называются числами с плавающей запятой (floating-point). Имя функции fabs является сокращением от англ. floating-point absolute value.)

Еще одним примером стандартной функции является pow из библиотеки с заголовочным файлом cmath. Эта функция выполняет возведение в степень. Так, чтобы поместить в переменную result значение x^y, можно выполнить такой оператор:

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

Следующие три строки программного кода выведут на экран число 9.0, поскольку результатом вычисления выражения $(3.0)^{2.0}$ будет значение 9.0:

```
double result, x = 3.0, y = 2.0;
result = pow(x, y);
cout << result;
```

Обратите внимание, что приведенный выше вызов функции pow возвращает значение 9.0, а не 9, поскольку она всегда возвращает значение типа double, а не int. Также заметьте, что данной функции требуются два аргумента, но в общем случае у функции может быть произвольное число аргументов. Каждый из них имеет строго определенный тип, и его значение, указываемое в вызове функции, должно принадлежать к этому типу. При задании аргумента не того типа C++, скорее всего, приведет его к нужному типу, если это возможно, однако результат может отличаться от того, который вы ожидали. Поэтому лучше всего всегда задавать значения в строгом соответствии с типами аргументов функции. Пожалуй, единственным исключением является автоматическое преобразование типа int к типу double. Как правило, на него можно полагаться и без опасений передавать значения типа int функциям, принимающим значения типа double.

Во многих реализациях функции pow существуют ограничения на использование аргументов. В частности, если первый аргумент функции pow отрицателен, второй должен быть целым числом. А поскольку в процессе изучения C++ у вас и так хватит забот, используйте функцию pow для возведения в степень только неотрицательных чисел.

Преобразование типов

Как вы уже знаете, операция 9/2 представляет собой целочисленное деление, и ее результатом является 4 а не 4.5. Если же нужно получить результат типа double (то есть число с дробной частью), то как минимум один из двух операндов должен иметь этот тип. Так, 9/2.0 равно 4.5. Когда одно из двух чисел задается в виде константы, достаточно просто добавить десятичную точку и нуль — результат получится с дробной частью.

Но как быть, если оба операнда, как показано в следующем примере, являются переменными?

```
int total_candy, number_of_people;
double candy_per_person;

... Программа помещает в переменную total_candy значение 9, а в переменную
    number_of_people значение 2. Как это делается, в данном случае не имеет значения. ...

candy_per_person = total_candy/number_of_people;
```

Если не привести значение одной из переменных, total_candy или number_of_people, к типу double, результатом деления будет 4, а не 4.5. И то, что переменная candy_per_person имеет тип double, ничего не изменит. Полученное в результате деления значение 4 будет преобразовано к типу double и помещено в переменную candy_per_person — только и всего! Иными словами, значение 4 будет преобразовано в 4.0 и переменной candy_per_person будет присвоено значение 4.0, а не 4.5. Если бы одним из операндов операции деления была константа, можно было бы добавить к ней точку и нуль, но мы же имеем дело с двумя переменными. Однако выход есть, поскольку в C++ предусмотрен способ преобразования значений типа int к типу double, применимый и к константам и к переменным. Например, вот так записывается указание «преобразовать значение 9 к типу double»:

```
static_cast<double>(9)
```

Выражение static_cast<double> — это нечто вроде стандартной функции, преобразующей значение, такое как 9, к заданному типу данных, в нашем случае double. Данная операция называется *преобразованием типов* или *приведением типов*. Вместо double может быть задан другой тип данных, но более подробно мы поговорим об этом позже.

В следующем примере значение 9 типа int приводится к типу double и переменной answer присваивается значение 4.5.

```
double answer;
answer = static_cast<double>(9)/2;
```

Преобразование типов применимо также для констант и может использоваться, чтобы облегчить понимание кода, поскольку ясно показывает намерение программиста. Однако никаких новых возможностей в данном случае оно не дает. Если вам требуется значение 9 типа double, можно написать 9.0 или static_cast<double>(9.0) — результат будет одинаковым. Но когда в делении участвуют две переменные, преобразование необходимо. Попробуем переписать приведенный выше пример так, чтобы переменной candy_per_person присваивалось верное значение 4.5. Для этого вместо total_candy в последней строке напишем static_cast<double>(total_candy):

```
int total_candy, number_of_people;
double candy_per_person;

... Программа помещает в переменную total_candy значение 9, а в переменную number_of_people
    значение 2. Как это делается, в данном случае не имеет значения. ...

candy_per_person =
     static_cast<double>(total_candy)/number_of_people;
```

Обратите внимание на размещение скобок. Приведение типов выполняется раньше деления, потому что в противном случае цифры после десятичной точки будут потеряны. Если по ошибке написать строку

```
candy_per_person =
     static_cast<double>(total_candy/number_of_people); // НЕВЕРНО
```

переменная candy_per_person получит значение 4.0, а не 4.5.

Преобразование значения типа int к типу double

Конструкция static_cast<double> может использоваться в качестве стандартной функции, преобразующей заданное значение к типу double. Так, static_cast<double>(2) возвращает значение 2.0. Данная операция называется *преобразованием* или *приведением типов*. (Приведение можно выполнять не только к типу double, но пока мы ограничимся им, а о приведении к другим типам данных поговорим позднее.)

Синтаксис

```
static_cast<double>(выражение_типа_int)
```

Пример

```
int total_pot, number_of_winners;
double your_winnings;
   ...
your_winnings =
      static_cast<double>(total_pot)/number_of_winners;
```

Старая форма оператора приведения типов

В настоящее время приведение типов в C++ рекомендуется выполнять с помощью оператора static_cast<тип>. Однако в ранних версиях C++ использовался другой синтаксис: для приведения значения к некоторому типу задавалось имя этого типа, а за ним в скобках значение, как если бы имя типа было именем функции. Например, преобразование значения 9 к типу double выполнялось так:

```
double(9)
```

То есть, если переменная candy_per_person имеет тип double, а переменные total_candy и number_of_people — тип int, присваивание

```
candy_per_person =
      static_cast<double>(total_candy)/number_of_people;
```

эквивалентно

```
candy_per_person =
          double(total_candy)/number_of_people;
```

Но хотя выражения static_cast<double>(total_candy) и double(total_candy) более или менее равнозначны, рекомендуется использовать только первую форму, поскольку в будущих версиях C++ старая форма может не поддерживаться.

Ловушка: при целочисленном делении отбрасывается дробная часть

При выполнении целочисленного деления, например 11/2, помните, что результатом будет 5, а не 5.5. В результате такого деления всегда генерируется целое число, полученное путем округления частного к меньшему значению, совершенно независимо от последующего использования этого результата.

Рассмотрим пример:

```
double d;
d = 11/2;
```

В этом случае оба операнда являются целыми числами, следовательно, над ними выполняется целочисленное деление. Полученное в результате значение 5 преобразуется к типу double и присваивается переменной d, а дробная часть утрачивается. При этом тот факт, что переменная d имеет тип double, никак не отражается на результате деления — она все равно получает значение 5.0, а не 5.5.

Упражнения для самопроверки

1. Определите значение каждого из арифметических выражений.

sqrt(16.0)	sqrt(16)	pow(2.0, 3.0)
pow(2, 3)	pow(2.0, 3)	pow(1.1, 2)
abs(3)	abs(-3)	abs(0)
fabs(-3.0)	fabs(-3.5)	fabs(3.5)
ceil(5.1)	ceil(5.8)	floor(5.1)
floor(5.8)	pow(3.0, 2)/2.0	pow(3.0, 2)/2
7/abs(-2)	(7 + sqrt(4.0))/3.0	sqrt(pow(3, 2))

2. Преобразуйте следующие математические выражения в арифметические выражения на C++.

$\sqrt{x+y}$ $\quad x^{y+7}$ $\quad \sqrt{area+fudge}$

$\dfrac{\sqrt{time+tide}}{nobody}$ $\quad \dfrac{-b+\sqrt{b^2-4ac}}{2a}$ $\quad |x-y|$

3. Напишите завершенную программу на C++, вычисляющую и выводящую на экран значение квадратного корня из π (π приблизительно равно 3,14159). Воспользуйтесь соответствующей константой с именем M_PI, определенной в библиотеке cmath как const double M_PI.
4. Напишите и откомпилируйте небольшие программы, чтобы выяснить следующее:
   а) позволяет ли ваш компилятор размещать директиву #include <iostream> в любом месте строки или только в ее начале (без пробела перед символом #);
   б) допускает ли ваш компилятор пробелы между символом # и словом include.

3.3. Функции, определяемые программистом

Костюм, сшитый на заказ, всегда сидит лучше готового.

Мой дядя, портной

В предыдущем разделе мы рассмотрели, как используются готовые функции. Теперь поговорим о создании собственных функций.

Определения функций

Собственную функцию можно определить либо в том же файле, где находится главная часть программы (main), либо в отдельном файле для ее использования в нескольких разных программах.

В листинге 3.2 приведен пример определения функции total_cost в составе демонстрационной программы, которая вызывает эту функцию. Она принимает два аргумента — цену единицы товара и количество покупаемых единиц — и возвращает общую стоимость покупки, включая налог. Вызов функции total_cost выглядит точно так же, как вызов любой функции из стандартных библиотек. Однако ее описание, которое должен написать программист, несколько сложнее.

Листинг 3.2. Определение функции

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

// Объявление функции.
double total_cost(int number_par, double price_par);
// Вычисляет общую стоимость, включая  5 % налог, для
// number_par единиц товара, продаваемых по цене price_par.

int main()
{
    double price, bill;
    int number;

    cout << "Enter the number of items purchased: ";
    cin >> number;
    cout << "Enter the price per item $";
    cin >> price;

    // Вызов функции.
    bill = total_cost(number, price);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at "
         << "$" << price << " each.\n"
         << "Final bill, including tax, is $" << bill
         << endl;

    return 0;
}
// Начало определения функции.
double total_cost(int number_par, double price_par) // Заголовок функции.
// Начало тела функции.
{
    const double TAX_RATE = 0.05; // 5 % налог
    double subtotal;

    subtotal = price_par * number_par;
```

продолжение ⇩

Листинг 3.2 (*продолжение*)

```
    return (subtotal + subtotal*TAX_RATE);
}
// Конец тела функции.
// Конец определения функции.
```

Пример диалога

```
Enter the number of items purchased: 2
Enter the price per item $10.10
2 items at $10.10 each.
Final bill, including tax, is $21.21
```

Описание состоит из двух частей, называемых объявлением и определением функции. *Объявление функции* или *прототип функции* показывает, как ее вызывать. Язык C++ требует, чтобы перед вызовом функции в программе обязательно располагалось либо ее полное определение, либо объявление. Вот объявление функции total_cost из листинга 3.2:

```
double total_cost(int number_par, double price_par);
```

Объявление содержит все, что программисту нужно знать для написания вызова функции — в нем указано имя функции, в данном случае total_cost, и перечислены ее аргументы с указанием их типов. Здесь у функции total_cost два аргумента, первый имеет тип int, а второй — double. Идентификаторы number_par и price_par называются *формальными параметрами*. Формальный параметр указывает место, куда при вызове функции должен подставляться реальный аргумент, и используется для обозначения последнего при написании функции. Именами формальных параметров могут быть любые допустимые идентификаторы, но пока мы будем добавлять к ним суффикс _par, чтобы отличать их от других элементов программы. Заметьте, что объявление функции оканчивается точкой с запятой.

Первое слово в объявлении функции определяет тип возвращаемого ею значения, отсюда следует, что функция total_cost возвращает значение типа double.

Как видите, следующий вызов функции в листинге программы 3.2:

```
bill = total_cost(number, price);
```

полностью соответствует ее объявлению.

Вызов функции — это выражение справа от знака равенства, из которого ясно, что функция называется total_cost и у нее два аргумента. Первый имеет тип int, а второй — double, и поскольку переменная bill имеет тип double, похоже, что функция возвращает значение типа double. Так оно и есть на самом деле, что определяется объявлением функции.

Объявление функции

Объявление функции несет в себе всю информацию, необходимую программисту для ее вызова. Оно должно располагаться в программном коде до вызова функции, если ему не предшествует ее полное определение. Обычно объявления функций помещают перед текстом главной части программы.

Синтаксис

тип_возвращаемого_значения имя_функции(список_параметров);
комментарий_к_объявлению_функции

Здесь *список_параметров* — это разделенный запятыми список параметров:

тип_1 формальный_параметр_1, тип_2 формальный_параметр_2, ... ,
тип_последний формальный_параметр_последний

Пример

```
// Не забудьте точку с запятой в конце.
double total_waight(int number, double waight_of_one);
// Возвращает общий вес number элементов,
// каждый из которых весит waight_of_one.
```

Определение функции, помещенное в листинге 3.2 в самом конце, описывает процесс вычисления возвращаемого функцией значения. Если представить себе эту функцию как маленькую программу в программе, ее определение подобно коду этой маленькой программы. Фактически синтаксис определения функции очень схож с синтаксисом главной части программы. Оно включает заголовок функции, за которым следует тело функции. *Заголовок функции* имеет тот же синтаксис, что и ее объявление, — только в конце отсутствует точка с запятой. Таким образом, заголовок функции повторяется в программе дважды, но это не является ошибкой.

Хотя объявление функции содержит информацию, необходимую для ее вызова, в нем указан только тип возвращаемого значения и ничего не сказано о том, как это значение формируется. Процесс его формирования определяется операторами в *теле функции*. Тело функции следует за ее заголовком и завершает ее определение. Оно состоит из объявлений и последовательности исполняемых операторов, заключенных в фигурные скобки. Таким образом, тело функции имеет в точности такой же вид, как тело главной части программы. При вызове функции значения аргументов подставляются в формальные параметры, а затем выполняются операторы тела функции. Возвращаемое функцией значение определяется оператором `return`. (Подробнее о том, как выполняется подстановка значений аргументов функции, рассказывается далее в этом разделе.)

Оператор `return` состоит из ключевого слова `return`, за которым следует выражение. Определение функции в листинге 3.2 содержит такой оператор `return`:

```
return (subtotal + subtotal*TAX_RATE);
```

В результате его выполнения в качестве значения функции возвращается значение выражения

```
(subtotal + subtotal*TAX_RATE)
```

Использование скобок не обязательно. Следующий оператор `return` эквивалентен приведенному выше:

```
return subtotal + subtotal*TAX_RATE;
```

Однако оператор return, включающий более длинное выражение, легче читается при наличии скобок. Некоторые программисты убеждены, что ради единообразия программного кода даже самые простые выражения в этом операторе лучше помещать в скобках. В листинге программы 3.2 после оператора return нет больше никаких операторов, но если бы они присутствовали, то не были бы выполнены. После выполнения оператора return работа функции завершается.

Теперь посмотрим, что происходит при выполнении следующего вызова функции в листинге программы 3.2:

```
bill = total_cost(number, price);
```

Прежде всего, значения аргументов number и price подставляются в формальные параметры number_par и price_par. В примере диалога для программы из листинга 3.2 переменной number присваивается значение 2, а переменной price — значение 10.10, поэтому в параметры number_par и price_par подставляются значения 2 и 10.10. Эта процедура называется *передачей параметров по значению*, а формальные параметры именуют *параметрами, передаваемыми по значению*. Что касается процесса передачи параметров по значению, обратите внимание на следующее.

1. В формальные параметры подставляются значения аргументов. Если они являются переменными, в функцию передаются их значения, а не сами переменные.
2. Первый аргумент подставляется в первый формальный параметр из списка параметров функции, второй аргумент — во второй параметр и т. д.
3. При передаче аргумента в функцию его значение подставляется во все вхождения формального параметра в тело этой функции (например, каждый раз, когда в теле функции встречается параметр number_par, вместо него в нашем примере подставляется значение 2).

Функция подобна небольшой программе

Для понимания того, что такое функция и как она действует, нужно усвоить следующее:

- функцию можно сравнить с небольшой программой, а вызов функции сходен с вызовом этой программы;
- для ввода данных в функцию используются формальные параметры, а не оператор cin. Аргументы функции являются входными значениями и подставляются в формальные параметры;
- функция обычно не выводит ничего на экран, а вместо этого направляет нечто вроде «вывода» вызвавшей ее программе. Функция возвращает значение, что можно сравнить с выводом программой выходных данных, только вместо оператора cout она использует оператор return.

Процесс вызова функции в примере диалога для программы из листинга 3.2 показан на рис. 3.1.

Вызов функции в программе из листинга 3.2

1. Перед вызовом функции переменным number и price с помощью операторов cin присваиваются значения 2 и 10.10 (см. пример диалога к программе из листинга 3.2).

2. Начинается выполнение следующего оператора, содержащего вызов функции:

```
bill = total_cost(number, price);
```

3. Значение переменной number (то есть 2) подставляется в параметр number_par, а значение переменной price (то есть 10.10) в параметр price_par:

```
// Заголовок функции
double total_cost(int number_par, double price_par)
{
    const double TAX_RATE = 0.05; // 5 % налог
    double subtotal;

    subtotal = price_par * number_par;
    return (subtotal + subtotal*TAX_RATE);
}
```

и в результате получается:

```
// Заголовок функции
double total_cost(int 2, double 10.10)
{
    const double TAX_RATE = 0.05; // 5 % налог
    double subtotal;

    subtotal = 2 * 10.10;
    return (subtotal + subtotal*TAX_RATE);
}
```

4. Выполняется тело функции, то есть следующий код:

```
{
    const double TAX_RATE = 0.05; // 5 % налог
    double subtotal;

    subtotal = 2 * 10.10;
    return (subtotal + subtotal*TAX_RATE);
}
```

5. При выполнении оператора return функция возвращает значение выражения, стоящего после ключевого слова return. В данном случае при выполнении оператора

```
return (subtotal + subtotal*TAX_RATE);
```

вызов функции

```
total_cost(number, price)
```

возвращает значение (subtotal + subtotal*TAX_RATE), равное 21.21, в результате чего по окончании выполнения оператора

```
bill = total_cost(number, price);
```

перменная bill (слева от знака равенства) получает значение 21.21.

Рис. 3.1. Подробное описание процесса вызова функции

Две формы объявлений функций

В объявлении функции не обязательно задавать имена ее формальных параметров. Следующие два объявления:

```
double total_cost(int number_par, double price_par);
```

и

```
double total_cost(int, double);
```

эквивалентны.

В примерах данной книги мы используем первую форму, для того чтобы на формальные параметры можно было ссылаться в пояснительных комментариях к объявлениям функций. Однако в справочниках и руководствах часто встречается вторая форма объявления[1].

Использование сокращенной формы допустимо только в объявлении функции. В заголовке же обязательно должны быть перечислены имена всех ее формальных параметров.

Ловушка: неверный порядок аргументов

При вызове функции компьютер подставляет первый аргумент в первый формальный параметр, второй аргумент во второй формальный параметр и т. д., не проверяя аргументы на логическое соответствие параметрам (да он и не может этого сделать). Поэтому если перепутать порядок аргументов в вызове функции, программа просто выполнит не те действия, которые требуются. Чтобы понять, каковы могут быть последствия, посмотрите на программу, приведенную в листинге 3.3. Написавший ее программист по ошибке переставил аргументы в вызове функции grade. Ее вызов должен был бы иметь такой вид:

```
letter_grade = grade(score, need_to_pass);
```

Это единственная ошибка в программе, определяющей оценку студента, но она может привести к тому, что какой-нибудь учащийся не сдаст экзамен. В данном случае функция grade настолько проста, что при ее тестировании программист наверняка обнаружит ошибку, однако в более сложной функции подобная ошибка может остаться незамеченной.

Если тип аргумента не соответствует типу формального параметра, компилятор может выдать предупреждающее сообщение, но это делают не все компиляторы. Более того, в такой ситуации, как в нашем примере, когда оба формальных параметра имеют один и тот же тип данных, неверный порядок аргументов не приводит к несоответствию типов, и ни один компилятор не выдаст предупреждения.

[1] Все, что нужно C++ для компоновки используемой в программе библиотеки или функции, — это имя функции и типы ее параметров. Имена формальных параметров требуются только в определении функции. Однако программа должна быть понятна не только компилятору, но и программисту, а имена параметров часто несут информацию об их назначении.

Листинг. 3.3. Неверный порядок аргументов

```
// Определяет оценку студента.
// Их две: Pass (сдал) и Fail (не сдал).
#include <iostream>
using namespace std;

char grade(int received_par, int min_score_par);
// Возвращает 'P', означающее Pass, если значение параметра
// received_par больше или равно min_score_par.
// В противном случае возвращает 'F' (Fail).

int main()
{
    int score, need_to_pass;
    char letter_grade;

    cout << "Enter your score"
         << " and the minimum needed to pass:\n";
    cin >> score >> need_to_pass;

    letter_grade = grade(need_to_pass, score);

    cout << "You received a score of "
         << score << endl
         << "Minimum to pass is " << need_to_pass
         << endl;

    if (letter_grade == 'P')
        cout << "You Passed. Congratulations!\n";
    else
        cout << "Sorry. You failed.\n";

    cout << letter_grade
         << " will be entered in your record.\n";

    return 0;
}

char grade(int received_par, int min_score_par)
{
    if (received_par >= min_score_par)
        return 'P';
    else
        return 'F';
}
```

Пример диалога

```
Enter your score and the minimum needed to pass:
98 60
You received a score of 98
```

```
Minimum to pass is 60
Sorry. You failed.
F will be entered in your record.
```

Синтаксис определения функции

Объявления функций в программе обычно размещаются перед главной частью, а их определения – после нее (или же, как будет показано далее в этой книге, в отдельном файле). На рис. 3.2 приведен полный синтаксис объявления и определения функции. Практически этот синтаксис дает несколько большую свободу, чем показано на рисунке. Объявления переменных и констант могут чередоваться в определении функции с исполняемыми операторами, но каждая переменная должна быть объявлена до ее использования. Правила смешения объявлений и исполняемых операторов в определении функции те же, что и для главной части программы. Однако без веских причин этого лучше не делать, а размещать все объявления единым блоком в начале функции, как на рис. 3.2.

Поскольку функция не возвращает значения до выполнения оператора `return`, в ее теле должен присутствовать хотя бы один такой оператор (но их может быть и несколько). Например, в теле функции может присутствовать оператор `if...else` с оператором `return` в каждой ветви, как в листинге 3.3.

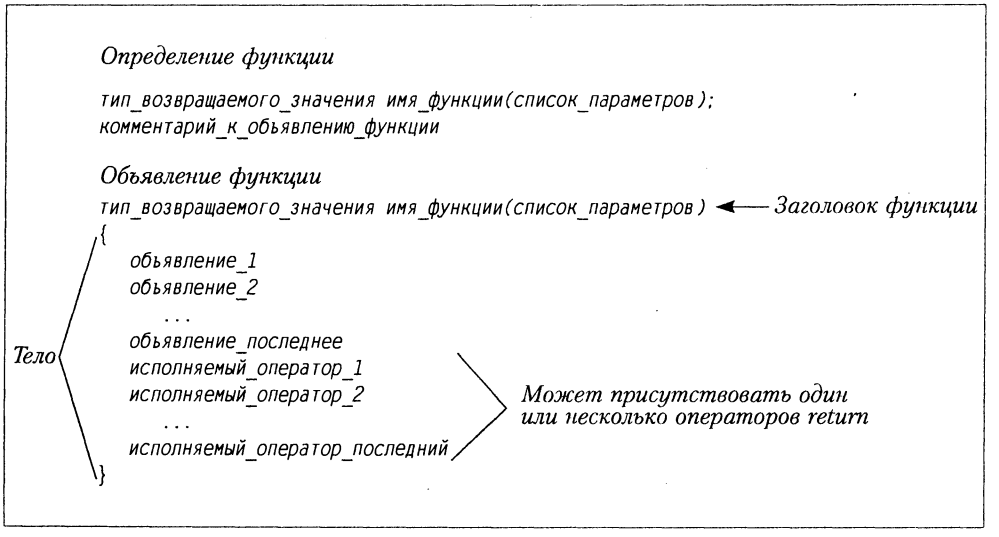


Рис. 3.2. Синтаксис функции, возвращающей значение

В теле функции компилятор примет любое разумное расположение пробелов и разрывов строк. Но чтобы достичь согласованности, элементы функции следует размещать так же, как в главной части программы. Обратите особое внимание на расположение фигурных скобок в примерах и на рис. 3.2. Скобки, в которые заключено тело функции, должны находиться на отдельных строках для четкого визуального обозначения его границ.

Еще об определениях функций

Мы рассказывали о том, где обычно располагаются объявления и определения функций, и предложили их оптимальное и общепринятое расположение. Но компилятор допускает и иные возможности. Правило, которому он следует, таково: вызову функции обязательно должно предшествовать ее определение или объявление. Например, если перед главной частью программы поместить определения всех функций, их объявления вообще не потребуются. Знание общего правила необходимо для понимания программ, с которыми вам придется столкнуться в будущем, однако мы советуем придерживаться рекомендаций, приведенных в данной книге (и иллюстрируемых примерами). Эти рекомендации описывают общепринятый стиль программирования и построения библиотек функций, которого придерживается большинство программистов, пишущих на C++.

Упражнения для самопроверки

5. Что выведет следующая программа:

```
#include <iostream>
using namespace std;
char mystery(int first_par, int second_par);
int main()
{
    cout << mystery(10, 9) << "ow\n";
    return 0;
}

char mystery(int first_par, int second_par)
{
    if (first_par >= second_par)
        return 'W';
    else
        return 'H';
}
```

6. Запишите объявление и определение функции, которая принимает три аргумента типа int и возвращает сумму их значений.
7. Запишите объявление и определение функции, которая принимает один аргумент типа int и один аргумент типа double и возвращает значение типа double, являющееся их средним арифметическим.
8. Запишите объявление и определение функции, которая принимает один аргумент типа double. Функция должна возвращать значение 'P', если ее аргумент положителен, и значение 'N', если он равен нулю или отрицателен.
9. Подробно опишите механизм передачи параметров по значению.
10. В чем сходство и в чем отличие между применением готовой (библиотечной) и пользовательской функций?

3.4. Процедурная абстракция

Причина скрыта, но результат хорошо известен.

Овидий

Аналогия с черным ящиком

Пользователя программы не интересуют подробности ее внутренней структуры. Представьте, что вам нужно было бы знать и помнить программный код компилятора! У каждой программы своя задача — откомпилировать заданную программу или проверить правописание определенного текста, или сделать что-нибудь еще. Вы пользуетесь множеством программ и знаете, какую задачу выполняет каждая из них, но обычно понятия не имеете, каким образом она ее выполняет. То же самое можно сказать и об используемой программистом функции — ведь, как мы уже говорили, функция подобна маленькой программе. Программист знает, что делает эта функция (вычисляет квадратный корень или преобразует значения температуры по Фаренгейту в значения по Цельсию), но не знает, как она это делает. Говорят, что программист воспринимает функцию как *черный ящик*.

Термин «черный ящик» происходит от метафорического образа механизма, помещенного в черный непрозрачный ящик с выведенными наружу средствами управления. Оператор умеет им управлять, но не знает, как он устроен. По аналогии в программировании черным ящиком называют любую систему с известным поведением и неизвестной внутренней структурой. Хорошо спроектированной функцией можно пользоваться как черным ящиком: если передать ей любой корректный набор аргументов, она всегда возвратит верный результат. Разработка функции в соответствии с этим принципом называется *сокрытием информации*. Программист может работать с такой функцией даже при условии, что ее тело скрыто.

В листинге 3.4 приведено объявление функции `new_balance` и два разных варианта ее определения. Как сказано в комментарии, функция вычисляет новый баланс банковского счета после прибавления начисленных процентов. Так, если сумма вклада составляла $100 и на нее начислено 4,5 %, новый баланс составит $104,50. Следующий код присваивает переменной `vacation_fund` значение `100.00`, а затем заменяет его значением `104.50`:

```
vacation_fund = 100.00;
vacation_fund = new_balance(vacation_fund, 4.5);
```

Листинг 3.4. Определения функции, эквивалентные с точки зрения использования ее как черного ящика

```
double new_balance(double balance_par, double rate_par);
// Возвращает баланс банковского счета
// после начисления процентов.
// Формальный параметр balance_par представляет собой
// исходный баланс.
// Формальный параметр rate_par представляет собой
// процентную ставку.
// Например, если rate_par равен 5.0, к балансу прибавляется
// 5 % и new_balance(100, 5.0) возвращает 105.00.
```

Определение 1

```
double new_balance(double balance_par, double rate_par);
{
    double interest_fraction, interest;

    interest_fraction = rate_par/100;
    interest = interest_fraction*balance_par;
    return (balance_par + interest);
}
```

Определение 2

```
double new_balance(double balance_par, double rate_par);
{
    double interest_fraction, updated_balance;

    interest_fraction = rate_par/100;
    updated_balance = balance_par*(1 + interest_fraction);
    return updated_balance;
}
```

Не имеет значения, какой из приведенных в листинге вариантов определения функции используется. При любых входных значениях они возвращают один и тот же результат, так что программисту вообще не нужно знать, как реализована функция, достаточно прочитать ее объявление и сопутствующий комментарий.

Принцип разработки и использования функции как черного ящика называют также *процедурной абстракцией*. В C++, скорее, стоило бы называть его функциональной абстракцией, но термин «процедура» более распространен, чем термин «функция». В информатике *процедурой* называют любую последовательность инструкций, подобную функциям C++. Что касается термина «абстракция», то в него заложен следующий смысл: когда функция используется как черный ящик, программист абстрагируется от кода, составляющего ее тело. В этом суть *принципа черного ящика*, *принципа процедурной абстракции* или *сокрытия информации*. Все эти три термина означают одно и то же. И какой бы из них вы не предпочли, важно понять основу этого принципа и следовать ему при разработке собственных функций.

Процедурная абстракция

Следование принципу *процедурной абстракции* при создании определения функции означает, что функция должна быть написана так, чтобы ею можно было пользоваться как *черным ящиком*. Другими словами, у применяющего ее программиста не должно быть потребности в просмотре тела функции для выяснения того, что она делает и как с ней работать.

Правила написания определения функции, которой можно пользоваться как черным ящиком.

- Комментарий к объявлению функции должен сообщать программисту всю необходимую информацию о ее аргументах и возвращаемом ею значении.
- Все используемые в теле функции переменные должны объявляться в нем же (формальные параметры не нуждаются в объявлении, поскольку они описаны в объявлении функции).

Совет программисту: выбор имен формальных параметров

Принцип процедурной абстракции гласит, что функция должна представлять собой модуль, разрабатываемый отдельно от остальной части программы. В больших проектах разные функции могут быть даже написаны разными программистами. Для формальных параметров функций следует выбирать предельно информативные имена. Как вы уже знаете, аргументами, подставляемыми в эти параметры, могут быть переменные из главной части программы — их имена тоже должны быть информативными. Поскольку имена аргументов и параметров часто придумывают разные программисты, вполне допустимо, что они могут совпасть. Но какими бы ни были имена переменных, используемых в качестве аргументов, они не будут конфликтовать с именами формальных параметров. Это очевидно еще и потому, что в функции используются не переменные, а их значения — имена этих переменных функция вообще игнорирует.

Теперь, когда вы знаете, что в выборе имен формальных параметров вам предоставлена полная свобода, можете больше не добавлять в их конец окончание _par. В качестве примера в листинге 3.5 приведено несколько модифицированное определение функции total_cost листинга программы 3.2, и формальные параметры number_par и price_par названы просто number и price.

Листинг 3.5. Упрощенные имена формальных параметров

Объявление функции

```
double total_cost(int number, double price);
// Вычисляет общую стоимость, включая  5 % налог,
// для number_par единиц товара, продаваемого по цене price_par.
```

Определение функции

```
// Заголовок функции
double total_cost(int number, double price)
{
    const double TAX_RATE = 0.05; // 5 % налог
    double subtotal;

    subtotal = price * number;
    return (subtotal + subtotal*TAX_RATE);
}
```

Пример: покупка пиццы

Упаковки или товары большого размера не всегда выгодны, это хорошо видно на примере с пиццей. Ее размер определяется диаметром в дюймах. Количественной мерой пиццы будем условно считать ее площадь, величина которой не пропорциональна диаметру. Большинство людей затруднятся определить разность площадей десятидюймовой и двенадцатидюймовой пицц, и поэтому им трудно решить, какую из них выгодней купить, то есть какая из них имеет меньшую цену в пересчете на квадратный дюйм. Выполним упражнение, в котором разработаем

программу, сравнивающую цену двух пицц и определяющую, какого размера пиццу выгоднее купить.

Постановка задачи

Точная спецификация входных и выходных данных программы такова.

Входные данные. Входными данными являются диаметры сравниваемых пицц.

Выходные данные. Выходные данные — стоимость квадратного дюйма каждой пиццы и сообщение, какую из них выгодней купить.

Анализ задачи

Согласно принципу нисходящего проектирования разделим программу на следующие подзадачи.

Подзадача 1. Получить входные данные для меньшей и большей пиццы.

Подзадача 2. Вычислить цену квадратного дюйма меньшей пиццы.

Подзадача 3. Вычислить цену квадратного дюйма большей пиццы.

Подзадача 4. Определить, какую пиццу выгоднее приобрести.

Подзадача 5. Вывести результаты.

Обратите внимание на подзадачи 2 и 3. У них имеются две важных особенности.

1. Это фактически одна и та же задача. Единственное различие состоит в том, что вычисления выполняются над разными данными. Иными словами, в подзадачах 2 и 3 различаются только размер и стоимость пиццы.
2. Результатом каждой из подзадач 2 и 3 является единственное значение — цена квадратного дюйма пиццы.

Когда подзадача принимает некоторые значения (например, числа) и возвращает единственное значение, ее лучше всего реализовать в виде функции. И если две или более подобных подзадач выполняют одни и те же вычисления, они могут быть реализованы как одна функция, вызываемая с разными аргументами. Итак, для вычисления цены квадратного дюйма пиццы напишем функцию, которую назовем `unitprice`. Ее объявление и сопутствующий комментарий будут такими:

```
double unitprice(int diameter, double price);
// Возвращает цену квадратного дюйма пиццы.
// Формальный параметр diameter
// представляет собой диаметр пиццы в дюймах.
// Формальный параметр price представляет стоимость пиццы.
```

Разработка алгоритма

Подзадача 1 проста. Программа запрашивает входные значения и сохраняет их в четырех переменных, названных `diameter_small`, `diameter_large`, `price_small` и `price_large`.

Подзадача 4 элементарна. Чтобы определить, какую пиццу выгоднее купить, нужно сравнить цены квадратного дюйма каждой из них с помощью оператора «меньше». Подзадача 5 — это вывод результатов.

Подзадачи 2 и 3 реализуются как вызовы функции unitprice. Затем разрабатывается ее алгоритм. Его самая сложная часть — определение площади пиццы. Зная площадь, можно определить цену квадратного дюйма с помощью простого деления:

```
price/area
```

Здесь area — переменная, представляющая значение площади пиццы, а price — параметр функции, где задается ее стоимость. Функция возвращает результат вычисления этого выражения. Однако нам нужно сформулировать метод вычисления площади пиццы.

По форме пицца представляет собой круг, его площадь (и соответственно площадь пиццы) равна πr^2, где r — радиус круга (равный половине диаметра), а π — число, приблизительно равное 3,14159.

Далее описан алгоритм функции unitprice. Общая его схема такова.

1. Вычислить радиус пиццы.
2. Вычислить площадь пиццы по формуле πr^2.
3. Вернуть значение выражения (price/area).

Перед переводом этого алгоритма на C++ мы детализируем его, представив в виде так называемого псевдокода. *Псевдокод* — это специальный язык для записи алгоритмов, который включает в себя как элементы языка программирования, так и фразы естественного языка. Такая запись позволяет точно выразить алгоритм, не заботясь о тонкостях синтаксиса C++, к тому же псевдокод легко преобразовывается в программу на C++. В нашем псевдокоде radius и area будут переменными для хранения значений радиуса и площади.

Псевдокод функции unitprice:

```
radius = половине диаметра;
area = π * radius * radius;
return (price/area);
```

На этом разработка алгоритма функции unitprice завершается. Теперь мы готовы к преобразованию подзадач 1–5 в полноценную программу на языке C++.

Кодирование

Поскольку о программировании ввода данных (подзадача 1) мы с вами не раз уже говорили, сразу перейдем к подзадачам 2 и 3. Их в нашей программе можно реализовать с помощью следующих двух вызовов:

```
unitprice_small = unitprice(diameter_small, price_small);
unitprice_large = unitprice(diameter_large, price_large);
```

Здесь unitprice_small и unitprice_large — переменные типа double. Одним из преимуществ функций является то, что одну и ту же функцию можно вызывать в программе неограниченное число раз. Это избавляет программиста от необходимости многократно повторять один и тот же (или почти один и тот же) код в тексте программы.

В результате преобразования нашего псевдокода на C++ получается вот такое тело функции unitprice:

```
// Первый черновой вариант тела функции unitprice.
{
   const double PI = 3.14159;
   double radius, area;

   radius = diameter/2;
   area = PI * radius * radius;
   return (price/area);
}
```

Заметьте, что с помощью квалификатора const мы определили PI как именованную константу. Кроме того, обратите внимание на следующую строку приведенного кода:

```
radius = diameter/2;
```

Это всего лишь деление на 2 — что может быть проще? Однако данная строка содержит серьезную ошибку. Ведь искомый радиус пиццы должен включать дробную часть, полученную в результате деления. Например, радиус 13-дюймовой пиццы составляет 6,5 дюйма. Однако переменная diameter имеет тип int, как и константа 2. Поэтому, как вы помните из главы 2, приведенный выше оператор выполнит целочисленное деление, результатом которого будет не 6.5, а 6. Эта ошибка вполне может остаться незамеченной, и миллионы членов Союза Потребителей Пиццы будут зря тратить деньги, покупая пиццу не того размера. А главное, программа не выполнит свою задачу — помочь клиенту выбрать наиболее подходящую покупку. Для более серьезной программы результаты подобной ошибки могут быть катастрофическими.

Как же ее исправить? Нам нужно, чтобы деление на 2 выполнялось как обычное, а не целочисленное, то есть результат должен содержать дробную часть. Для этого требуется хотя бы один операнд типа double. В данном случае можно привести к этому типу значение 2. Напомним, что оператор static_cast<double>(2), называемый оператором приведения типа, преобразует целочисленное значение 2 к типу double. Таким образом, если в названном выше операторе деления заменить 2 оператором static_cast<double>(2), первый операнд тоже будет преобразован к типу double и мы получим требуемый результат. Вот как будет выглядеть оператор деления после этой замены:

```
radius = diameter/static_cast<double>(2);
```

Полный код окончательного определения функции unitprice вместе с остальной частью программы показан в листинге 3.6.

Оператор приведения типа static_cast<double>(2) возвращает значение 2.0, так что вместо него можно было бы просто написать 2.0. Однако наша версия более информативна, поскольку показывает, какой метод деления мы намерены использовать. Если вместо static_cast<double>(2) просто написать 2.0, впоследствии при отладке или сопровождении программы это значение может быть случайно заменено значением 2, что приведет к только что описанной трудноуловимой ошибке.

И еще одно замечание относительно написания кода нашей программы. Как видно из листинга 3.6, кодируя задачи 4 и 5, мы объединили их в один блок кода, состоящий из последовательности операторов cout, за которыми следует оператор if...else. К такому объединению часто прибегают, когда две задачи очень просты и тесно связаны между собой.

Листинг 3.6. Покупка пиццы

```
// Определяет, пицца какого размера из двух
// предложенных выгоднее для покупки.
#include <iostream>
using namespace std;

double unitprice(int diameter, double price);
// Возвращает цену квадратного дюйма пиццы. Формальный
// параметр diameter представляет диаметр пиццы в дюймах.
// Формальный параметр price представляет стоимость пиццы в долларах.

int main()
{
    int diameter_small, diameter_large;
    double price_small, unitprice_small,
           price_large, unitprice_large;

    cout << "Welcome to the Pizza Consumers Union.\n";
    cout << "Enter diameter of a"
         << "small pizza (in inches): ";
    cin >> diameter_small;
    cout << "Enter the price of a small pizza: $";
    cin >> price_small;
    cout << "Enter diameter of a"
         << " large pizza (in inches): ";
    cin >> diameter_large;
    cout << "Enter the price of a large pizza: $";
    cin >> price_large;

    unitprice_small = unitprice(diameter_small, price_small);
    unitprice_large = unitprice(diameter_large, price_large);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Small pizza:\n"
         << "Diameter = " << diameter_small
         << " inches\n"
         << "Price = $" << price_small
         << " Per square inch = $" << unitprice_small
         << endl
         << "Large pizza:\n"
         << "Diameter = " << diameter_large
         << " inches\n"
         << "Price = $" << price_large
         << " Per square inch = $" << unitprice_large
         << endl;

    if (unitprice_large < unitprice_small)
        cout << "The large one is the better buy.\n";
```

```
    else
        cout << "The small one is the better buy.\n";
    cout << "Buon Appetito!\n";

    return 0;
}

double unitprice(int diameter, double price)
{
    const double PI = 3.14159;
    double radius, area;

    radius = diameter/static_cast<double>(2);
    area = PI * radius * radius;
    return (price/area);
}
```

Пример диалога

```
Welcome to the Pizza Consumers Union.
Enter diameter of a small pizza (in inches): 10
Enter the price of a small pizza: $7.50
Enter diameter of a large pizza (in inches): 13
Enter the price of a large pizza: $14.75
Small pizza:
Diameter = 10 inches
Price = $7.50 Per square inch = $0.10
Large pizza:
Diameter = 13 inches
Price = $14.75 Per square inch = $0.11
The small one is the better buy.
Buon Appetito!
```

Тестирование программы

То, что программа компилируется и выводит ответ, который кажется правильным, еще не означает, что она действительно работает корректно. Чтобы убедиться в отсутствии ошибок, нужно протестировать ее с входными значениями, для которых заранее известны правильные ответы (вычисленные на бумаге или с помощью калькулятора). Например, вряд ли кто-то станет покупать двухдюймовую пиццу, но для тестирования это значение вполне годится, поскольку удобно для вычислений вручную. Давайте посчитаем цену квадратного дюйма двухдюймовой пиццы, стоимостью $3,14. Ее радиус равен 1 дюйму, а площадь составляет $3{,}14159*1^2$, то есть 3,14159. Разделив значение, соответствующее стоимости пиццы, на значение площади мы получим цену квадратного дюйма, примерно равную $1,00. Конечно, это неправдоподобный размер и нереальная цена, но легко проверить, что для таких входных значений программа возвращает правильный результат.

После проверки программы с такими входными данными вы можете в какой-то мере ей доверять, но все же нельзя быть уверенным, что она работает полностью корректно. Неверно работающая программа иногда выдает правильные ответы, ошибаясь в расчетах с какими-нибудь другими входными данными. Может, вам просто повезло и вы тестировали ее с теми значениями, которые она обрабатывает

правильно. Предположим, что в программе допущена рассмотренная ранее ошибка, то есть вместо

```
radius = diameter/static_cast<double>(2);
```

написано

```
radius = diameter/2;
```

Задавая на входе четные значения диаметра, такие как 2, 8, 10 или 12, мы будем получать правильные ответы. Но если ввести нечетное число, программа выдаст неверный результат. Поэтому ее нужно тестировать, используя разные значения размера (и четные, и нечетные), при этом вероятность обнаружения имеющихся в ней ошибок увеличится.

Совет программисту: использование псевдокода

Алгоритмы часто записываются на языке, называемом псевдокодом. *Псевдокод* – это специальный язык для записи алгоритмов, который включает в себя как элементы языка программирования, так и фразы естественного языка (например, русского). Такая запись позволяет точно выразить алгоритм, не заботясь о тонкостях синтаксиса C++, к тому же псевдокод легко преобразовывается в программу на C++. Когда некоторый шаг алгоритма просто и понятно записать на C++, используйте для этого операторы языка программирования. В противном случае, его легче описать, используя конструкции естественного языка. В рассмотренном выше примере приводился псевдокод, применяемый для записи алгоритма функции `unitprice`.

Упражнения для самопроверки

11. Каково назначение комментария, сопровождающего объявление функции?
12. В чем заключается принцип процедурной абстракции в отношении определения функции?
13. Что имеется в виду, когда говорят, что программист может использовать функцию как черный ящик? (*Подсказка.* Этот вопрос очень тесно связан с предыдущим.)
14. Подробно опишите процесс тестирования программы.
15. Рассмотрите два возможных определения функции `unitprice`. Одно из них приведено в листинге 3.6. Второе очень похоже на предыдущее, но в нем выражение `static_cast<double>(2)` заменено константой `2.0`. Иными словами, вместо строки

    ```
    radius = diameter/static_cast<double>(2);
    ```

    записана строка

    ```
    radius = diameter/2.0;
    ```

    Эквивалентны ли эти два определения функции с точки зрения принципа черного ящика?

3.5. Локальные переменные

Он местный парень,
и за переделами своего городка никому не известен.

Распространенное высказывание

В предыдущем разделе мы объяснили, почему функции следует использовать по принципу черного ящика. Для этого они обычно должны содержать собственные переменные, а не обращаться к тем, которые объявлены во внешней программе. Такие переменные называются локальными. В этом разделе рассказывается, что собой представляют локальные переменные и как с ними работать.

Аналогия с маленькой программой

Посмотрите еще раз на листинг программы 3.1. В ней вызывается стандартная функция `sqrt`. Нам не нужно знать, как она реализована, и в частности, неважно, какие в ней объявляются переменные. В этом смысле от нее ничем не отличается и функция, определяемая программистом, где тоже могут объявляться переменные для локального использования. Причем, если объявить переменную в функции и еще одну с таким же именем в главной программе, это будут две разные переменные. Давайте рассмотрим пример такой программы.

Программа, приведенная в листинге 3.7, содержит две переменные с именем `average_pea`, одна объявляется и используется в определении функции `est_total`, а вторая – в главной части программы. Это две разные переменные, и они не конфликтуют друг с другом, как не конфликтовали бы переменные в двух абсолютно разных программах. Когда в функции `est_total` переменной `average_pea` присваивается новое значение, то значение одноименной переменной в главной части программы не изменяется. (О тех деталях листинга программы 3.7, которые не связаны с совпадением имен переменных, рассказывается далее в разделе «Пример: расчет предполагаемого урожая».)

Листинг 3.7. Локальные переменные

```
// Вычисляет среднее значение урожая гороха
// на экспериментальном участке.
#include <iostream>
using namespace std;

double est_total(int min_peas, int max_peas, int pod_count);
// Возвращает оценочное общее количество собранных горошин.
// В формальном параметре pod_count
// задается количество стручков.
// В формальных параметрах min_peas
// и max_peas задается минимальное
// и максимальное количество горошин в стручке.

int main()
{
   int max_count, min_count, pod_count;
```

продолжение ⇩

Листинг 3.7 (*продолжение*)

```
    double average_pea, yield;   // Эта переменная локальна для главной части программы.

    cout << "Enter minimum and maximum"
         << " number of peas in a pod: ";
    cin >> min_count >> max_count;
    cout << "Enter the number of pods: ";
    cin >> pod_count;
    cout << "Enter the weight"
         << " of an average pea (in ounces): ";
    cin >> average_pea;

    yield = est_total
      (min_count, max_count, pod_count) * average_pea;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(3);
    cout << "Min number of peas per pod = "
         << min_count << endl
         << "Max number of peas per pod = "
         << max_count << endl
         << "Pod count = " << pod_count << endl
         << "Average pea weight = "
         << average_pea << " ounces" << endl
         << "Estimated average yield = "
         << yield << " ounces"
         << endl;

    return 0;
}

double est_total(int min_peas, int max_peas, int pod_count)
{
    double average_pea;   // Эта переменная локальна для функции est_total.
    average_pea = (max_peas + min_peas)/2.0;
    return (pod_count * average_pea);
}
```

Пример диалога

```
Enter minimum and maximum number of peas in a pod: 4 6
Enter the number of pods: 10
Enter the weight of an average pea (in ounces): 0.5
Min number of peas per pod = 4
Max number of peas per pod = 6
Pod count = 10
Average pea weight = 0.500 ounces
Estimated average yield = 25.000 ounces
```

Переменные, объявленные в теле функции, называются *локальными переменными этой функции*, а функция — *областью их видимости*. Переменные, объявленные в главной части программы, именуются *локальными для главной части программы*, которая является *областью их видимости*. Существуют и другие виды переменных, не являющихся локальными ни для одной функции или главной части программы. Когда переменную называют *локальной* без упоминания о функции

или главной части программы, имеется в виду, что она локальна для какой-либо из функций.

Локальные переменные

Переменные, объявленные в теле функции, называются *локальными переменными этой функции*, а функция — *областью их видимости*. Переменные, объявленные в главной части программы, именуются *локальными для главной части программы*, которая является *областью их видимости*. Когда переменную называют *локальной* без упоминания о функции или главной части программы, имеется в виду, что она локальна для какой-либо функции. Если переменная локальна для функции, в главной части программы или в другой функции можно создать переменную с таким же именем, и это будут две разные переменные.

Пример: расчет предполагаемого урожая

Программа из листинга 3.7 оценивает урожай гороха, собранный с маленького экспериментального участка. Используемая в ней функция `est_total` возвращает количество собранных горошин. У этой функции три аргумента: в первом задается количество собранных стручков, а второй и третий нужны для оценки среднего количества горошин в стручке. Стручки содержат разное количество горошин, и во втором и третьем аргументах задается минимальное и максимальное количество горошин в стручке. Функция вычисляет среднее арифметическое этих двух чисел и использует его как среднее количество горошин в стручке.

Глобальные константы и глобальные переменные

В главе 2 мы уже говорили, что для определения используемых в программе констант применяется квалификатор `const`. Например, в листинге 3.6 использовалось следующее определение идентификатора `PI`, представляющего собой константу 3,14159:

```
const double PI = 3.14159;
```

В листинге 3.2 с помощью квалификатора `const` таким же образом определялась процентная ставка налога:

```
const double TAX_RATE = 0.05; // Налог, составляющий 5 %.
```

Объявления констант, как и объявления переменных, мы поместили в тело функций, где они используются. Это вполне удовлетворяет нашим требованиям, поскольку каждая константа применяется только в одной функции. Однако часто бывает так, что одна и та же константа используется в нескольких функциях. Тогда можно поместить ее объявление в начало программы вне тел всех функций и вне тела главной части программы. Такая константа называется *глобальной именованной константой* и может применяться в определениях всех функций, следующих за ее объявлением.

В листинге 3.8 приведен пример программы, содержащей глобальную именованную константу. Данная программа запрашивает у пользователя радиус и вычисляет площадь соответствующего круга и объем соответствующей сферы. В геометрии существуют для этого стандартные формулы:

площадь = $\pi \times (радиус)^2$
объем = $(4/3) \times \pi \times (радиус)^3$

Напомним, что константа π приблизительно равна 3,14159. Ранее мы использовали для нее следующее определение:

```
const double PI = 3.14159;
```

В листинге программы 3.8 идентификатор `PI` объявляется точно так же, но не в функции, а в начале файла, как глобальная именованная константа, что позволяет использовать его во всех функциях.

Компилятор очень терпим в отношении размещения объявлений глобальных именованных констант, но для того чтобы код был читабельным, лучше всего размещать однотипные операторы вместе: в одной группе все директивы `#include`, в другой все объявления глобальных констант, в третьей все объявления функций. Следуя общепринятой практике, мы будем записывать объявления констант после директив `#include` и перед объявлениями функций.

Размещение объявлений именованных констант в начале программы помогает сделать ее более читабельной, даже если константа используется только в одной функции. Когда в какой-нибудь из будущих версий программы значение константы потребуется изменить, ее легко будет найти. Например, если процент налога задать в начале программы, то в случае изменения его значения модифицировать эту константу будет проще.

Переменные, объявляемые без квалификатора `const`, тоже могут быть *глобальными*, то есть доступными во всех определенных в файле функциях. Для этого их, как и глобальные константы, нужно объявить в начале файла вне тела всех функций и главной части программы. Однако нужда в таких глобальных переменных возникает редко, более того, их использование может затруднить понимание и сопровождение программы, поэтому в данной книге мы ими пользоваться не будем. Но со временем по мере освоения языка C++ вы иногда будете прибегать и к этой возможности.

Листинг 3.8. Глобальная именованная константа

```
// Вычисляет площадь круга и объем шара.
// В обоих вычислениях используется одно и то же значение радиуса.
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159;

double area(double radius);
// Возвращает площадь круга с заданным радиусом.

double volume(double radius);
// Возвращает объем шара с заданным радиусом.
```

```
int main()
{
    double radius_of_both, area_of_circle, volume_of_sphere;

    cout << "Enter a radius to use for both a circle\n"
         << "and a sphere (in inches): ";
    cin >> radius_of_both;

    area_of_circle = area(radius_of_both);
    volume_of_sphere = volume(radius_of_both);

    cout << "Radius = " << radius_of_both << " inches\n"
         << "Area of circle = " << area_of_circle
         << " square inches\n"
         << "Volume of sphere = " << volume_of_sphere
         << " cubic inches\n";

    return 0;
}

double area(double radius)
{
    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    return ((4.0/3.0) * PI * pow(radius, 3));
}
```

Пример диалога

```
Enter a radius to use for both a circle
and a sphere (in inches): 2
Radius = 2 inches
Area of circle = 12.5664 square inches
Volume of sphere = 33.5103 cubic inches
```

Формальные параметры, передаваемые по значению, являются переменными

Формальные параметры являются не просто обозначениями мест для подстановки значений аргументов функции, на самом деле — это локальные переменные функции, которые могут использоваться точно так же, как любые другие локальные переменные. Ранее в данной главе описывался механизм передачи параметров по значению, теперь мы остановимся на этом вопросе подробнее. При вызове функции ее формальные параметры (являющиеся ее локальными переменными) инициализируются значениями аргументов. Именно в этом заключается точный смысл «подстановки значений в параметры», о которой говорилось ранее. Как правило, формальный параметр используется только в качестве вместилища для значения аргумента, но функция может и изменять его значение. Далее будет приведен пример формального параметра, используемого в качестве полноценной локальной переменной.

Программа из листинга 3.9 генерирует счета за юридические консультации для фирмы Девей, Чизема и Хаува. Заметьте, что в отличие от других юридических фирм, здесь предусмотрена особая тарификация: время консультации делится на 15-минутные промежутки, и если она длится менее четверти часа, оплата не берется. Например, когда консультация продолжается час и четырнадцать минут, счет выставляется за четыре четверти часа, а не за пять, как в других фирмах, поэтому клиент заплатит всего $600, а не $750.

Обратите внимание на формальный параметр minutes_worked в определении функции fee. Он используется как переменная, и его значение изменяется так:

```
minutes_worked = hours_worked*60 + minutes_worked;
```

Формальные параметры являются такими же переменными, как локальные переменные, объявленные в теле функции. Однако объявлять их отдельно не нужно, поскольку для этого служит список формальных параметров в заголовке определения функции. Таким образом, следующий фрагмент определения функции fee:

```
double fee(int hours_worked, int minutes_worked)
{
    int quarter_hours;
    int minutes_worked; // Так поступать нельзя.
...
```

неверен, поскольку в нем параметр minutes_worked объявляется дважды.

Листинг 3.9. Формальные параметры, используемые в качестве локальных переменных

```
// Программа, «выписывающая» счета за юридические консультации.
#include <iostream>
using namespace std;

const double RATE = 150.00; // Долларов за четверть часа.

double fee(int hours_worked, int minutes_worked);
// Возвращает сумму оплаты за hours_worked часов
// и minutes_worked минут юридической консультации.

int main()
{
    int hours, minutes;
    double bill;

    cout << "Welcome to the offices of\n"
         << "Dewey, Cheatham, and Howe.\n"
         << "The law office with a heart.\n"
         << "Enter the hours and minutes"
         << " of your consultation:\n";
    cin >> hours >> minutes;

    bill = fee(hours, minutes);
    // Значение переменной minutes не изменяется функцией fee


    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "For " << hours << " hours and " << minutes
```

```
        << " minutes, your bill is $" << bill << endl;

    return 0;
}

double fee(int hours_worked, int minutes_worked)
// minutes_worked - это локальная переменная, инициализируемая значением
// переменной minutes при вызове fee(hours, minutes) в программе.
{
    int quarter_hours;

    minutes_worked = hours_worked*60 + minutes_worked;
    quarter_hours = minutes_worked/15;
    return (quarter_hours*RATE);
}
```

Пример диалога

```
Welcome to the offices of
Dewey, Cheatham, and Howe.
The law office with a heart.
Enter the hours and minutes of your consultation:
2 45
For 2 hours and 45 minutes, your bill is $1650.00
```

Пространство имен std

До сих пор мы начинали все программы следующей парой строк:

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

Однако начало файла — не всегда лучшее место для строки

```
using namespace std;
```

Позже мы будем использовать не только std, но и другие пространства имен, и возможно, даже различные пространства имен для разных функций. Если поместить директиву

```
using namespace std;
```

в тело функции (внутрь фигурных скобок, в которые оно заключено), то она будет относиться только к определению данной функции. Это позволяет в определениях разных функций использовать различные пространства имен, даже если функции определены в одном файле и некоторые имена из разных пространств имен совпадают, хотя каждая функция имеет при этом специфическое назначение для своего конкретного пространства имен.

Размещение директивы using в определении функции сходно с размещением в нем объявления переменной. Переменная, объявленная в теле функции, является для нее локальной. Иначе говоря, действие объявления ограничивается определением функции. Точно так же и действие директивы using, помещенной в определении функции, ограничивается этим определением.

И хотя пока мы не будем использовать другие пространства имен, кроме std, привыкайте размещать директивы using там, где они используются.

В листинге 3.10 приведена та же программа, что и в листинге 3.8, но директивы using здесь размещены в определениях функций. В данном случае разница заключается только в стиле программирования, а в остальном программы полностью эквивалентны. Но при использовании нескольких пространств имен различие станет принципиальным.

Листинг 3.10. Использование пространств имен

```
// Вычисляет площадь круга и объем сферы.
// В обоих вычислениях используется одно и то же значение радиуса.
#include <iostream>
#include <cmath>

const double PI = 3.14159;

double area(double radius);
// Возвращает площадь круга с заданным радиусом.

double volume(double radius);
// Возвращает объем сферы с заданным радиусом.

int main()
{
    using namespace std;

    double radius_of_both, area_of_circle, volume_of_sphere;

    cout << "Enter a radius to use for both a circle\n"
         << "and a sphere (in inches): ";
    cin >> radius_of_both;

    area_of_circle = area(radius_of_both);
    volume_of_sphere = volume(radius_of_both);

    cout << "Radius = " << radius_of_both << " inches\n"
         << "Area of circle = " << area_of_circle
         << " square inches\n"
         << "Volume of sphere = " << volume_of_sphere
         << " cubic inches\n";

    return 0;
}

double area(double radius)
{
    using namespace std;

    return (PI * pow(radius, 2));
}

double volume(double radius)
{
    using namespace std;

    return ((4.0/3.0) * PI * pow(radius, 3));
}
```

Упражнения для самопроверки

16. Где следует объявлять переменную, используемую в определении функции: в определении функции, в главной части программы, в любом удобном месте?
17. Предположим, что в теле функции `Function1` объявлена переменная `Sam`, и переменная с таким же именем объявлена в теле функции `Function2`. Будет ли откомпилирована такая программа (предположим, что все остальное в ней правильно)? Если да, запустится ли она и будут ли при ее выполнении выдаваться сообщения об ошибках (считаем, что в остальном программа правильна)? Если программа будет выполнена, причем без сообщений об ошибках, выдаст ли она правильные результаты (считаем, что в остальном программа правильна)?
18. Следующая функция должна принимать в качестве аргументов длину, выраженную в футах и дюймах, и возвращать общее количество дюймов, содержащихся в заданном количестве футов и дюймов. Так, вызов `total_inches(1, 2)` должен вернуть 14, поскольку 1 фут и 2 дюйма равны 14 дюймам (напомним, что в 1 футе содержится 12 дюймов). Будет ли следующая функция:

```
double total_inches(int feet, int inches)
{
    inches = 12*feet + inches;
    return inches;
}
```

правильно выполнять свою задачу?

19. Напишите объявление и определение функции с именем `read_filter`, не имеющей параметров и возвращающей значение типа `double`. Функция предлагает пользователю ввести значение данного типа и считывает его в локальную переменную, а затем возвращает либо значение, если оно оказывается большим или равным нулю, либо нуль, если значение оказывается отрицательным.

Пример: функция, вычисляющая факториал

Листинг 3.11 содержит объявление и определение функции, выполняющей известную математическую операцию — вычисление факториала. В математике факториал числа n обозначается как $n!$ (читается «n факториал») и вычисляется следующим образом:

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

В определении функции мы будем выполнять умножение в цикле `while`. Обратите внимание, что в программе умножение удобнее выполнять в обратном порядке: сначала на n, потом на $n-1$, затем на $n-2$ и т. д. Определение функции `factorial` содержит две локальные переменные: `product`, которая объявлена в начале тела функции, и формальный параметр `n`. Так как формальный параметр является локальной переменной, его значение можно изменять. В данном случае мы будем делать это с помощью оператора декрементирования `n--` (о нем говорилось в главе 2).

Листинг 3.11. Функция, вычисляющая факториал

Объявление функции

```
int factorial (int n);
// Возвращает n факториал.
// Аргумент n не должен быть отрицательным.
```

Определение функции

```
int factorial (int n)
{
    int product = 1;
    while (n > 0)
    {
        product = n * product;
        n--;   // Формальный параметр n используется как локальная переменная.
    }
    return product;
}
```

При каждом выполнении цикла значение переменной product умножается на значение переменной n, после чего значение переменной n с помощью оператора n-- уменьшается на единицу. Если функция factorial вызывается с аргументом 3, при первой итерации цикла переменная product получает значение 3, затем значение 3*2, а потом значение 3*2*1, после чего цикл while завершается. Поэтому оператор

```
x = factorial(3);
```

присваивает переменной x значение 6 (равное 3*2*1).

Заметьте, что локальная переменная product инициализируется значением 1 прямо в объявлении. Чтобы понять, почему выбрано именно это значение, вспомните, что после первой итерации цикла переменная product должна быть равна значению формального параметра.

3.6. Перегрузка имен функций

– Значит, так: триста шестьдесят четыре дня в году
ты можешь получать подарки на день нерождения.
– Совершенно верно, – сказала Алиса.
– И только один раз на день рождения! Вот тебе и слава!
– Я не понимаю, при чем здесь «слава»? – спросила Алиса.
Шалтай-Болтай презрительно улыбнулся.
– И не поймешь, пока я тебе не объясню, – ответил он.
– Я хотел сказать: «Разъяснил, как по полкам разложил!».
– Но «слава» совсем не значит «разъяснил,
как по полкам разложил!», – возразила Алиса.
– Когда я беру слово, оно означает то, что я хочу,
не больше и не меньше, – сказал Шалтай презрительно.
– Вопрос в том, подчинится ли оно вам, – сказала Алиса.
– Вопрос в том, кто из нас здесь хозяин, – сказал Шалтай-Болтай.

Льюис Кэрролл

Язык C++ позволяет задать два или более определений одного и того же имени функции, благодаря чему имя с одинаковой семантикой допустимо использовать

в разных ситуациях. Например, можно задать три функции с именем max: одна будет определять большее из двух чисел, вторая – большее из трех чисел, а еще одна – большее из четырех чисел. Создание двух или более определений функции называется *перегрузкой* имени функции. Злоупотреблять этой возможностью не стоит, поскольку перегрузка требует особой тщательности и внимания при разработке и сопровождении функций, однако в некоторых случаях ее применение очень эффективно.

Основные понятия перегрузки

Предположим, что вы пишете программу, в которой должно вычисляться среднее арифметическое двух чисел. Для выполнения такой операции можно использовать следующую функцию:

```
double ave(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}
```

Теперь предположим, что в той же программе требуется вычисление среднего арифметического трех чисел. Для данной операции можно написать такую функцию:

```
double ave3(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

Это вполне эффективное решение, а во многих других языках программирования у вас вообще не будет иного выбора. Однако C++ позволяет решить задачу более изящно – назвать обе функции одним и тем же именем. Иными словами, вместо определения функции ave3 можно написать такое определение:

```
double ave(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

и в результате у функции ave будет два определения, что и является примером перегрузки этой функции.

В листинге 3.12 два определения функции ave приведены в составе полной программы. Обратите внимание, что каждому из них соответствует свое объявление.

Перегрузка – замечательная вещь, так как делает программу более читабельной. Она избавляет от необходимости каждый раз придумывать для функции новое имя только потому, что наиболее естественное имя уже использовано в определении другой ее версии. Но как же компилятор узнает, какое из определений применять, когда в тексте программы встречается обращение к имени функции, уже имеющей два определения? Компилятор просто сравнивает количество и типы аргументов с определениями функций и выбирает то из них, в котором они совпадут. В листинге 3.12 одна из функций ave имеет два аргумента, а вторая – три. Для того чтобы решить, какую из них использовать, компилятор подсчитывает количество

аргументов в вызове функции: если их два, используется первая функция, а если три — вторая.

Когда одно и то же имя функции имеет несколько определений, они должны различаться спецификациями аргументов. Это означает, что никакая пара одноименных функций не должна иметь одинаковых наборов аргументов, совпадающих и по количеству и по типам данных. Обратите внимание, что различаться должны именно параметры: перегрузка функции не может основываться только на различии типов значений, возвращаемых ее реализациями.

Перегрузка имени функции

Наличие двух или более определений одного имени функции называется *перегрузкой*. При перегрузке имени функции ее определения должны отличаться количеством и/или типами формальных параметров. Во время вызова функции компилятор использует то ее определение, в котором набор формальных параметров соответствует набору передаваемых функции аргументов.

Листинг 3.12. Перегрузка имени функции

```
// Демонстрирует перегрузку имени функции ave.
#include <iostream>

double ave(double n1, double n2);
// Возвращает среднее арифметическое двух чисел, n1 и n2.

double ave(double n1, double n2, double n3);
// Возвращает среднее арифметическое трех чисел, n1, n2 и n3.

int main()
{
    using namespace std;
    cout << "The average of 2.0, 2.5, and 3.0 is "
         << ave(2.0, 2.5, 3.0) << endl;

    cout << "The average of 4.5 and 5.5 is "
         << ave(4.5, 5.5) << endl;

    return 0;
}

double ave(double n1, double n2) // Два аргумента.
{
    return ((n1 + n2)/2.0);
}

double ave(double n1, double n2, double n3) // Три агрумента.
{
    return ((n1 + n2 + n3)/3.0);
}
```

Пример диалога

```
The average of 2.0, 2.5, and 3.0 is 2.5
The average of 4.5 and 5.5 is 5
```

На самом деле концепция перегрузки для вас не нова. В главе 2 рассказывалось о перегрузке оператора деления /. Если оба операнда имеют тип int, этот оператор выполняет целочисленное деление. Например, 13/2 равно 6. Если же хоть один из операндов имеет тип double, он выполняет обычное деление и результат содержит дробную часть. Так, 13/2.0 равно 6.5. Таким образом, эти два определения оператора деления различаются типами операндов. Различие между перегрузкой оператора деления и перегрузкой имен пользовательских функций заключается только в том, что во втором случае перегрузку приходится программировать самостоятельно. Далее в этой книге вы узнаете, как перегружать другие операторы, такие как +, - и т. п.

Пример: модифицированная программа покупки пиццы

Союз Потребителей Пиццы успешно пользуется программой, приведенной в листинге 3.6. Теперь каждый может осуществлять такую покупку, которая для него наиболее выгодна. Один нечестный продавец пытался убеждать потребителей покупать более дорогую пиццу, но наша программа положила конец его махинациям. Однако производители нашли новый способ морочить головы покупателям: теперь они предлагают не только круглую, но и прямоугольную пиццу, зная, что наша программа работает только с круглой. Но мы тоже не будем отставать и доработаем свою программу, чтобы разрушить их коварные планы. Наша новая программа должна уметь сравнивать круглую пиццу с прямоугольной.

Изменения, которые необходимо внести в программу, очевидны: нужно несколько изменить ее ввод и вывод для поддержки двух разных форм пиццы. Кроме того, необходимо добавить новую функцию, которая будет вычислять цену квадратного дюйма прямоугольной пиццы. Можно определить ее так:

```
double unitprice_rectangular
       (int length, int width, double price)
{
   double area = length * width;
   return (price/area);
}
```

Однако имя этой функции настолько длинное, что нам даже пришлось разбить ее заголовок на две строки. Ничего неправильного в этом нет, но гораздо удобнее было бы называть обе функции, вычисляющие цену квадратного дюйма пиццы, одним и тем же именем. А поскольку C++ допускает перегрузку имен, это вполне возможно. Наличие двух определений имени функции unitprice не приведет к каким-либо проблемам с компилятором, поскольку в них указано различное число аргументов. Программа, сравнивающая пиццы разной формы, приведена в листинге 3.13.

Листинг 3.13. Перегрузка имени функции

```
// Определяет, какая пицца выгоднее для покупки:
// круглая или прямоугольная.
#include <iostream>

double unitprice(int diameter, double price);
// Возвращает цену квадратного дюйма круглой пиццы. Формальный
// параметр diameter представляет диаметр пиццы в дюймах.
// Формальный параметр price представляет стоимость пиццы в долларах.

double unitprice(int length, int width, double price);
// Возвращает цену квадратного дюйма прямоугольной
// пиццы размером length на width дюймов.
// Формальный параметр price представляет стоимость пиццы в долларах.

int main()
{
    using namespace std;
    int diameter, length, width;
    double price_round, unit_price_round,
           price_rectangular, unitprice_rectangular;

    cout << "Welcome to the Pizza Consumers Union.\n";
    cout << "Enter the diameter in inches"
         << " of a round pizza: ";
    cin >> diameter;
    cout << "Enter the price of a round pizza: $";
    cin >> price_round;
    cout << "Enter length and width in inches\n"
         << "of a rectangular pizza: ";
    cin >> length >> width;
    cout << "Enter the price of a rectangular pizza: $";
    cin >> price_rectangular;

    unitprice_rectangular =
                unitprice(length, width, price_rectangular);
    unit_price_round = unitprice(diameter, price_round);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);

    cout << endl
         << "Round pizza: Diameter = "
         << diameter << " inches\n"
         << "Price = $" << price_round
         << " Per square inch = $" << unit_price_round
         << endl
         << "Rectangular pizza: Length = "
         << length << " inches\n"
         << "Rectangular pizza: Width = "
         << width << " inches\n"
         << "Price = $" << price_rectangular
         << " Per square inch = $"
         << unitprice_rectangular
         << endl;

    if (unit_price_round < unitprice_rectangular)
```

```
        cout << "The round one is the better buy.\n";
    else
        cout << "The rectangular one is the better buy.\n";
    cout << "Buon Appetito!\n";

    return 0;
}

double unitprice(int diameter, double price)
{
    const double PI = 3.14159;
    double radius, area;

    radius = diameter/static_cast<double>(2);
    area = PI * radius * radius;
    return (price/area);
}

double unitprice(int length, int width, double price)
{
    double area = length * width;
    return (price/area);
}
```

Пример диалога

```
Welcome to the Pizza Consumers Union.
Enter the diameter in inches of a round pizza: 10
Enter the price of a round pizza: $8.50
Enter length and width in inches
of a rectangular pizza: 6 4
Enter the price of a rectangular pizza: $7.55

Round pizza: Diameter = 10 inches
Price = $8.50 Per square inch = $0.11
Rectangular pizza: Length = 6 inches
Rectangular pizza: Width = 4 inches
Price = $7.55 Per square inch = $0.31
The round one is the better buy.
Buon Appetito!
```

Автоматическое преобразование типов

Предположим, что в программе имеется следующее определение функции mpg:

```
double mpg(double miles, double gallons)
// Возвращает количество миль на галлон.
{
    return (miles/gallons);
}
```

и имя этой функции не перегружено, то есть в программе определена только одна функция с таким именем.

Если вызвать данную функцию с аргументами типа int, C++ автоматически приведет их к типу double. Поэтому следующий вызов:

```
cout << mpg(45, 2) << " miles per gallon";
```

выведет на экран текст 22.5 miles pre gallon. C++ преобразует значение 45 в 45.0, а 2 в 2.0, а затем выполнит деление 45.0/2.0 и получит 22.5.

Если функции требуется аргумент типа double, а при вызове ей передается аргумент типа int, C++ автоматически преобразует его к типу double. Это настолько естественно, что мы об этом даже не задумываемся. Однако перегрузка имен функций вносит в данный процесс свои коррективы.

Предположим, что мы непродуманно перегрузили имя функции mpg следующим образом:

```
int mpg(int goals, int misses)
// Возвращает число забитых голов,
// равное (goals - misses).
{
    return (goals - misses);
}
```

В программе, содержащей оба эти определения, следующий вызов:

```
cout << mpg(45, 2) << " miles per gallon";
```

выведет 43 miles per gallon (поскольку 43 равно 45 – 2)

Когда C++ встречает вызов mpg(45, 2) с двумя аргументами типа int, он сначала ищет определение функции mpg с двумя формальными параметрами типа int. После того как компилятор находит такое определение, оно используется в вызове, и никакого преобразования типов не происходит.

Этот пример демонстрирует еще один важный нюанс перегрузки: одно и то же имя ни в коем случае не следует использовать для двух не связанных между собой функций, поскольку это приводит к серьезным логическим ошибкам.

Упражнения для самопроверки

20. Предположим, что у нас имеются два определения имени функции score со следующими объявлениями:

    ```
    double score(double time, double distance);
    ```

    и

    ```
    int score(double points);
    ```

    Какое из определений будет использоваться в вызове

    ```
    final_score = score(x);
    ```

    и почему (переменная x имеет тип double)?

21. Предположим, что у нас имеются два определения имени функции the_answer со следующими объявлениями:

    ```
    double the_answer(double data1, double data2);
    ```

    и

    ```
    double the_answer(double time, int count);
    ```

    Какое из определений будет использоваться в вызове

    ```
    x = the_answer(y, 6.0);
    ```

    и почему (переменные x и y имеют тип double)?

22. Предположим, что у нас имеются два определения имени функции the_answer, приведенных в упражнении 21. Какое из них будет использоваться в вызове

```
x = the_answer(5, 6);
```

и почему?

23. Предположим, что у нас имеются два определения имени функции the_answer, приведенных в упражнении 21. Какое из них будет использоваться в вызове

```
x = the_answer(5, 6.0);
```

и почему?

24. Данный вопрос связан с примером программирования «Модифицированная программа покупки пиццы». Предположим, что недобросовестный продавец, постоянно пытающийся навязать покупателям более дорогие товары, добавил в ассортимент квадратную пиццу. Можно ли перегрузить функцию unitprice, чтобы она могла вычислять цену одного квадратного дюйма пиццы квадратной формы? Поясните ответ.

25. Посмотрите на листинг программы 3.13. Функция main содержит такую директиву using:

```
using namespace std;
```

Почему функция unitprice не содержит такую же директиву?

Резюме

- Наилучшая стратегия разработки алгоритма программы заключается в том, чтобы разбить выполняемую программой задачу на несколько подзадач, каждую из них — на меньшие подзадачи и т. д., пока не получатся настолько простые подзадачи, что их легко будет реализовать на C++. Этот подход называется нисходящим проектированием.
- Функция, возвращающая значение, подобна маленькой программе. Ее аргументы исполняют роль входных данных «программы», а возвращаемое значение — роль ее выходных данных.
- Когда подзадача программы принимает несколько входных значений и выдает одно выходное, ее можно реализовать в виде функции.
- Функция должна быть определена так, чтобы ею можно было пользоваться как черным ящиком. Программиста не интересуют детали ее программного кода, ему нужно только объявление функции с сопутствующим комментарием, описывающим ее аргументы и возвращаемое значение. Это правило называют принципом процедурной абстракции.
- Переменная, объявленная в определении функции, называется локальной для этой функции.
- Глобальные именованные константы объявляются с помощью квалификатора const. Их объявления обычно размещаются в начале программы после директив #include и до объявлений функций.

- Формальные параметры, передаваемые по значению (единственный описанный в этой главе тип формальных параметров), являются переменными, локальными для данной функции. Иногда удобно использовать формальные параметры именно в таком качестве, то есть изменять их значения.
- Наличие двух или более определений одного имени функции называется перегрузкой. При перегрузке имени функции все ее определения должны отличаться количеством и/или типами формальных параметров.

Ответы к упражнениям для самопроверки

1.
```
4.0     4.0     8.0
8.0     8.0     1.21
3       3       0
3.0     3.5     3.5
6.0     6.0     5.0
5.0     4.5     4.5
3       3.0     3.0
```

2.
```
sqrt (x + y)                pow(x, y + 7)                   sqrt(area + fudge)
sqrt(time + tide)/nobody    (-b + sqrt(b*b - 4*a*c)) /(2*a) abs(x - y) или
                                                            labs(x - y) или
                                                            fabs(x - y)
```

3.
```
// Вычисляет квадратный корень из 3.14159.
#include <iostream>
#include <cmath>   // Содержит sqrt и M_PI.
using namespace std;

int main()
{
    cout << "The square root of " << M_PI << " is "
         << sqrt(M_PI) << endl;
    return 0;
}
```

4. а)
```
// Определяет, поддерживает ли компилятор
// пробелы перед знаком # в записи #include.
#include <iostream>
using namespace std;

int main()
{
    cout << "hello world" << endl;
    return 0;
}
```

б)
```
// Определяет, поддерживает ли компилятор
// пробелы между знаком # и словом include в записи #include.
# include <iostream>
using namespace std;
// Остальная часть программы может быть такой же,
// как в предыдущем упражнении.
```

5. Wow

6. Объявление функции следующее:

```
int sum(int n1, int n2, int n3);
// Возвращает сумму n1, n2 и n3.
```

А определение такое:

```
int sum(int n1, int n2, int n3)
{
    return (n1 + n2 + n3);
}
```

7. Объявление функции следующее:

```
double ave(int n1, double n2);
// Возвращает среднее арифметическое n1 и n2.
```

А определение такое:

```
double ave(int n1, double n2)
{
    return ((n1 + n2)/2.0);
}
```

8. Объявление функции следующее:

```
char positive_test(double number);
// Возвращает 'P', если число положительно.
// Возвращает 'N', если число отрицательно или равно нулю.
```

А определение такое:

```
char positive_test(double number)
{
    if (number > 0)
        return 'P';
    else
        return 'N';
}
```

9. Предположим, что функция определена с двумя параметрами, param1 и param2. Она вызывается с соответствующими аргументами arg1 и arg2. Значения аргументов подставляются в соответствующие формальные параметры: arg1 в param1, а arg2 в param2. Затем эти значения используются в функции.

10. Для использования готовой (библиотечной) функции обычно нужно включить (с помощью директивы #include) в программу заголовочный файл. Что касается функции, определяемой программистом, ее код нужно добавить либо в тот же файл, где находится главная часть, либо в другой файл, компилируемый и компонуемый вместе с программой.

11. В комментарии указывается, какое значение возвращает функция, а также приводится другая информация, необходимая для ее использования.

12. Принцип процедурной абстракции гласит, что функция должна быть написана так, чтобы ею можно было пользоваться как черным ящиком. Это означает, что применяющему ее программисту не нужно просматривать тело функции,

чтобы узнать, что она делает и как с ней работать. Объявление функции и сопровождающий его комментарий — это все, что нужно для ее использования.

13. Когда говорят, что программист может использовать функцию как черный ящик, имеется в виду следующее: ему не нужно просматривать тело функции, чтобы узнать, что она делает и как с ней работать. Объявление функции и сопутствующий комментарий содержат всю информацию, необходимую для ее использования.

14. Для того чтобы убедиться, что программа работает корректно (по крайней мере, с большой вероятностью), нужно протестировать ее, используя входные данные, для которых известны правильные результаты. Ответы обычно вычисляют другими способами, например вручную на бумаге или с помощью калькулятора.

15. Да, с точки зрения принципа черного ящика оба определения имени функции эквивалентны, поскольку они возвращают одно и то же значение.

16. Если переменная используется в определении функции, там она и должна быть объявлена.

17. Программа откомпилируется, выполнится без сообщений об ошибках и выдаст правильные результаты.

18. Функция будет выполнять свою задачу правильно. В качестве комментария можно добавить, что значение аргумента, переданного этой функции по значению, после ее выполнения останется неизменным, хотя соответствующий формальный параметр используется функцией в качестве локальной переменной и его значение в ходе вычисления результирующего значения функции изменяется.

19. Объявление этой функции следующее:

```
double read_filter();
// Считывает число с клавиатуры. Возвращает это же число,
// если оно >=0, и нуль в противном случае.
```

А определение такое:

```
// Используем библиотеку классов iostream.
double read_filter();
{
    using namespace std;
    double value_read;
    cout << "Enter a number:\n"
    cin >> value_read;

    if (value_read >= 0)
        return value_read;
    else
        return 0.0;
}
```

20. В вызове функции задан только один аргумент, поэтому для его выполнения будет использоваться определение имени функции с одним формальным параметром.

21. В вызове функции заданы два аргумента типа double, поэтому для его выполнения будет использоваться определение имени функции с двумя аргументами типа double (то есть первое определение).
22. Второй аргумент, заданный в вызове функции, имеет тип int, как во втором определении функции. Поэтому для выполнения вызова будет использоваться второе определение, а первый аргумент будет автоматически преобразован к типу double.
23. Второй аргумент, заданный в вызове функции, имеет тип double, как в первом определении функции. Поэтому для выполнения вызова будет использоваться первое определение, а первый аргумент будет автоматически преобразован к типу double.
24. Этого сделать нельзя (по крайней мере, простым и естественным способом). Размер обоих видов пиццы определяется одинаково: целым числом, представляющим диаметр круглой пиццы или длину одной стороны квадратной пиццы. Поэтому и в том, и в другом случае функция unitprice будет иметь один формальный параметр типа double, представляющий стоимость пиццы, и один формальный параметр типа int, представляющий ее размер. Иными словами, количество и типы параметров в определениях функций будут совпадать, и компилятор не сможет решить, какое из них использовать в каждом конкретном случае. Разумеется, можно добавить в программу еще одну функцию с другим именем и таким способом решить задачу.
25. Функция unitprice не выполняет ни ввода, ни вывода, поэтому она не пользуется библиотекой iostream. В функции main директива using необходима, поскольку идентификаторы cin и cout определены в библиотеке iostream и отнесены к пространству имен std.

Практические задания

1. Напишите программу, которая будет запрашивать количество литров бензина, израсходованного автомобилем пользователя, и количество пройденных машиной миль. Программа должна выводить значение расстояния в милях, на преодоление которого машина тратит один галлон горючего, и позволять пользователю повторять вычисления необходимое количество раз. В ней следует использовать глобальную именованную константу, представляющую количество литров в галлоне (1 литр равен 0,264179 галлона).
2. Стоимость акции обычно округляется до одной восьмой доллара; например, 29 7/8 или 89 1/2. Напишите программу, вычисляющую и выводящую общую стоимость принадлежащих пользователю акций. Она запрашивает количество акций, целую часть цены акции в долларах и ее дробную часть. Последняя вводится как два значения типа int, одно представляет числитель, а второе — знаменатель. Необходимо, чтобы программа позволяла пользователю повторять вычисления желаемое количество раз. Она должна содержать определение функции с тремя аргументами типа int, представляющими целую

часть количества долларов, составляющего цену акции, а также числитель и знаменатель его дробной части. Функция возвращает цену акции в виде одного числа типа `double`.

3. Напишите программу, вычисляющую коэффициент инфляции прошлого года. Программа запрашивает цену товара (скажем, хот-дога или бриллианта в один карат) в настоящее время и год назад. Коэффициент инфляции равен частному от деления разности текущей и прошлогодней цены на прошлогоднюю цену. Программа должна позволять пользователю повторять вычисления желаемое количество раз. Определите функцию, вычисляющую коэффициент инфляции. Его нужно представить как значение типа `double` (например, `0.053`, которое соответствует инфляции в 5,3 %).

4. Дополните программу предыдущего задания, чтобы она выводила предполагаемую цену товара через год и два с момента вычислений. Вероятное увеличение стоимости товара за год равно коэффициенту инфляции, умноженному на текущую цену. Определите еще одну функцию, вычисляющую цену товара через год на основании текущей цены и коэффициента инфляции.

5. Напишите определение функции, вычисляющей сумму, начисленную по вкладу на кредитной карточке. Функция принимает в качестве аргументов начальный баланс, процент месячного начисления и количество месяцев, за которые произведены начисления. Не забудьте, что баланс на карточке, исходя из которого начисляется процент, увеличивается каждый месяц. Воспользуйтесь циклом `while`, похожим на цикл в листинге 2.7 (но не обязательно точно таким же). Включите эту функцию в программу, считывающую значение процента, начальный баланс и количество месяцев, а затем выводящую сумму начислений. Программа должна позволять пользователю повторять вычисления желаемое количество раз.

6. Сила гравитационного притяжения между двумя телами с массами m_1 и m_2, находящимися друг от друга на расстоянии d, определяют по формуле:

$$F = \frac{Gm_1m_2}{d^2}$$

где G – гравитационная постоянная:

$G = 6{,}673 \times 10^{-8}$ см^3/(г•с^2)

Напишите определение функции, принимающей в качестве аргументов массы двух тел и расстояние между ними и возвращающей значение силы их взаимного притяжения. В приведенной выше формуле сила измеряется в динах. Одна дина равна

г•см/с^2

Для гравитационной постоянной используйте глобальную константу. Включите определение функции в завершенную программу, вычисляющую силу притяжения между двумя телами на основе заданных данных. Программа должна позволять пользователю повторять вычисления желаемое количество раз.

7. Напишите программу, вычисляющую плату, которую необходимо внести за первый год владения купленным в кредит домом, с учетом налогообложения. Она вычисляется как величина годового платежа по кредиту минус сумма, сэкономленная за счет частичного освобождения от налогов. Программа должна запрашивать цену дома и сумму предварительного взноса. Величина годового платежа по кредиту составляет 3 % от начального размера ссуды (как возврат основного долга) плюс 8 % от размера непогашенной ссуды (как плата за пользование кредитом). Начальный размер ссуды рассчитывается как разность между ценой дома и суммой предварительного взноса. Предположим, что ставка налога равна 35 %, но платеж, представляющий собой плату за пользование кредитом, освобожден от налогообложения. Таким образом, сэкономленная на уплате налогов сумма составит 35 % от величины этого платежа. Программа должна использовать не менее двух определенных вами функций и предоставлять пользователю возможность выполнить вычисления необходимое количество раз.
8. Напишите программу, запрашивающую рост, вес и возраст человек и вычисляющую размер его одежды по следующим формулам.
   - Размер шляпы равен весу в фунтах, умноженному на 2,9 и деленному на рост в дюймах.
   - Размер пиджака (обхват груди в дюймах) равен весу, умноженному на 8 и деленному на 288 плюс 1/8 дюйма для каждых десяти лет возраста свыше 30. (Обратите внимание, что прибавление выполняется только после полных десяти лет. Поэтому для возраста от 30 до 39 лет не прибавляется ничего, а для возраста 40 лет прибавляется 1/8 дюйма.)
   - Обхват талии в дюймах равен весу, деленному на 5,7 плюс 1/10 дюйма для каждых двух лет возраста свыше 28. (Заметьте, что прибавление выполняется только после полных двух лет. Поэтому для возраста 29 не прибавляется ничего, а для возраста 30 лет прибавляется 1/10 дюйма.)

   Для каждого из вычислений используйте отдельную функцию. Программа должна позволять пользователю повторять вычисления желаемое количество раз.
9. Причины наличия в C++ нескольких стандартных функций для вычисления абсолютного значения носят исторический характер. Когда появился C++, уже существовали библиотеки для C, и никто не стал их переписывать с использованием перегрузки функций. Найдите все такие функции и перепишите их, перегружая имя функции `abs`. Как минимум у вас должны быть функции для значений типа `int`, `long`, `float` и `double`.

Глава 4 Функции для разных подзадач

Возможно все.
Известная максима

Стратегия нисходящего проектирования, рассмотренная в главе 3, является весьма эффективным способом разработки алгоритма программы. Согласно этой стратегии задача делится на подзадачи, алгоритмы которых реализуются в виде функций. До сих пор мы с вами имели дело с функциями, принимающими значения некоторого числа аргументов и возвращающими одно значение. Подзадачи, вычисляющие лишь одно значение, безусловно, очень важны, но это не единственный тип подзадач. В настоящей главе, завершая рассмотрение функций C++, мы опишем технологии разработки функций, выполняющих подзадачи других типов.

4.1. Функции типа void

В языке C++ подзадачи реализуются в виде функций. Функции, о которых рассказывалось в главе 3, всегда возвращают одно значение. Однако не все подзадачи можно реализовать с их помощью, поскольку существуют подзадачи, генерирующие несколько значений или не генерирующие их вовсе. В C++ функция должна возвращать либо одно значение, либо ни одного. Как вы увидите далее, подзадача, генерирующая несколько разных значений, обычно (как это ни парадоксально) реализуется в виде функции, не возвращающей значения. Для начала ограничимся задачами, которые не генерируют значений. Функция, не возвращающая значения, называется *функцией типа* `void`. Например, одной из типичных подзадач, выполняемых многими программами, является вывод на экран результатов каких-либо вычислений. Эта подзадача отображает данные на экране, но не формирует значения, передаваемого другой подзадаче. Поэтому ее можно реализовать как функцию типа `void`.

Определения функций типа void

В C++ определение функции типа void очень похоже на определение функции, возвращающей значение. В частности, приведенная ниже функция типа void выводит результат преобразования значений температуры по Цельсию в значения по Фаренгейту. Вычисления нас пока не интересуют – они выполняются в другой части программы, а функция только выводит результирующую информацию на экран.

```
void show_results(double f_degrees, double c_degrees)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << f_degrees
         << " degrees Fahrenheit is equivalent to\n"
         << c_degrees << " degrees Celsius.\n";
    return;
}
```

Как видно из этого примера, определение функции типа void имеет всего два отличия от определения функции, возвращающей значение (рассматривалась в главе 3). Во-первых, вместо типа возвращаемого функцией значения используется ключевое слово void, указывающее компилятору, что функция не возвращает ни одного значения. А во-вторых, оператор return не содержит выражения для возвращаемого значения. Полный синтаксис объявления и определения функции типа void представлен на рис. 4.1.

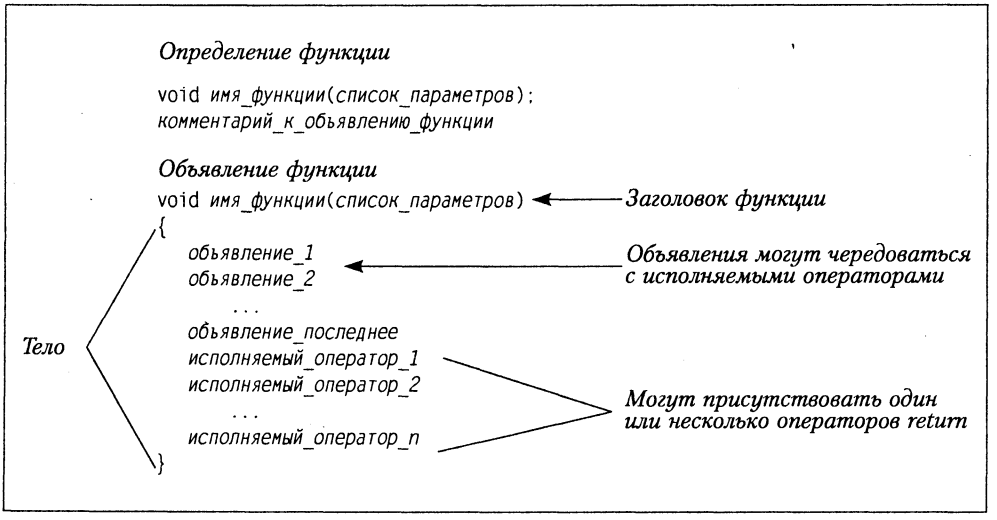


Рис. 4.1. Синтаксис определения функции типа void

Вызов функции типа void является исполняемым оператором. Например, приведенную выше функцию show_results можно вызвать так:

```
show_results(32.5, 0.3);
```

Обратите внимание, что вызов оканчивается точкой с запятой, указывающей компилятору, что это исполняемый оператор.

После выполнения в программе оператор выведет на экран следующее:

```
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

При вызове функции типа void аргументы подставляются в формальные параметры, а затем выполняются операторы тела функции. В частности, приведенный выше вызов функции show_results инициирует вывод на экран некоторой информации. Вызов функции типа void можно представить себе как копирование ее тела в программу на место оператора вызова (с предварительной подстановкой аргументов в параметры).

Использование функций без аргументов не только вполне допустимо, но часто очень полезно. Если функция не имеет аргументов, в ее объявлении отсутствует список формальных параметров и в ее вызове не задаются аргументы. Например, функция initialize_screen типа void выводит на экран символ перевода строки:

```
void initialize_screen()
{
    using namespace std;
    cout << endl;
    return;
}
```

Если в самом начале программы выполняется вызов

```
initialize_screen();
```

то выводимая этой программой информация всегда будет отделена от данных, отображаемых предыдущей программой, пустой строкой.

Заметьте, что даже если функция не имеет параметров, ее вызов и объявление должны содержать скобки. В следующем разделе описывается программа, включающая две функции типа void.

Пример: преобразование значений температуры

Программа, представленная в листинге 4.1, выполняет преобразование значения температуры по Фаренгейту в эквивалентное значение температуры по Цельсию, а результат выводит на экран. Вычисление производится по следующей формуле:

$C = (5/9)(F - 32)$

В программе, представленной в листинге 4.1, эту формулу реализует функция celsius.

Листинг 4.1. Функции типа void

```
// Программа преобразования значения температуры
// по Фаренгейту в значение температуры по Цельсию.
#include <iostream>

void initialize_screen();
```

```
// Отделяем пустой строкой данные, выводимые текущей программой,
// от данных, выводимых предыдущей.

double celsius(double fahrenheit);
// Преобразуем значение температуры по Фаренгейту
// в значение температуры по Цельсию.

void show_results(double f_degrees, double c_degrees);
// Выводим результаты. Предполагается, что значение переменной c_degrees (градусы
// по Цельсию) эквивалентно значению переменной f_degrees (градусы по Фаренгейту).

int main()
{
    using namespace std;
    double f_temperature, c_temperature;

    initialize_screen();
    cout << "I will convert a Fahrenheit temperature"
         << " to Celsius.\n"
         << "Enter a temperature in Fahrenheit: ";
    cin >> f_temperature;

    c_temperature = celsius(f_temperature);

    show_results(f_temperature, c_temperature);
    return 0;
}

// В определении используется класс iostream:
void initialize_screen()
{
    using namespace std;
    cout << endl;
    return;   // Этот оператор return необязателен.
}

double celsius(double fahrenheit)
{
    return ((5.0/9.0)*(fahrenheit - 32));
}

// В определении используется класс iostream:
void show_results(double f_degrees, double c_degrees)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << f_degrees
         << " degrees Fahrenheit is equivalent to\n"
         << c_degrees << " degrees Celsius.\n";
    return;   // Этот оператор return необязателен.
}
```

Пример диалога

```
I will convert a Fahrenheit temperature to Celsius.
Enter a temperature in Fahrenheit: 32.5
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

Оператор return в функциях типа void

И функции типа void, и функции, возвращающие значение, могут содержать оператор return. Однако если в первом случае этот оператор определяет возвращаемое функцией значение, то во втором он просто указывает на окончание вызова функции. Как упоминалось в предыдущей главе, каждая функция, возвращающая значение, должна завершаться исполняемым оператором return. А вот для функций типа void это не обязательно. Если данного оператора нет, выполнение функции завершается по окончании выполнения ее тела, как если бы перед конечной закрывающей фигурной скобкой стоял оператор return. Например так будут вести себя функции initialize_screen и show_results в программе, представленной в листинге 4.1, если удалить из них операторы return.

То, что перед закрывающей фигурной скобкой в конце определения функции типа void подразумевается оператор return, не означает, что этот оператор вообще не должен использоваться в функциях данного типа. Так, в функции, приведенной на рис. 4.2, он необходим. Эта функция подсказывает, как разделить заданное количество мороженого между посетителями ресторана. Если нет ни одного посетителя (что проверяется оператором if), выполнение функции тут же завершается во избежание деления на 0. Если же посетители есть, работа функции продолжается и выполняются операторы, следующие за оператором if.

Объявление функции

```
void ice_cream_division(int number, double total_weight);
// Нам необходимо разделить некоторое количество мороженого (total_weight)
// между всеми посетителями (number).
// Если значение переменной number равно 0, деление не выполняется.
```

Определение функции

```
// В определении используется класс iostream:
void ice_cream_division(int number, double total_weight)
{
     using namespace std;
     double portion;
     if (number == 0)
          return;  // Если значение переменной number равно 0,
                   // выполнение функции на этом завершается.
     portion = total_weight/number;
     cout.setf(ios::fixed);
     cout.setf(ios::showpoint);
     cout.precision(2);
     cout << "Each one receives "
          << portion << " ounces of ice cream." << endl;
}
```

Рис. 4.2. Использование оператора return в функции типа void

Вы, наверное, уже догадались, что главная часть программы — это определение функции main, которая автоматически активизируется при запуске программы и может, в свою очередь, вызывать другие функции. Хотя создается впечатление, что оператор return в функции main необязателен, на самом деле это не так. Функция main возвращает значение типа int, и поэтому ей необходим оператор return.

Однако используется она подобно функции типа void. В действительности, она не возвращает значение, и такое ее определение — лишь дань традиции. Лучше всего воспринимать указанную функцию как «главную часть программы» и не думать об этой незначительной детали[1].

Упражнения для самопроверки

1. Что выводит следующая программа:

```
#include <iostream>

void friendly();
void shy(int audience_count);


int main()
{
    using namespace std;
    friendly();
    shy(6);
    cout << "One more time:\n";
    shy(2);
    friendly();
    cout << "End of program.\n";
    return 0;
}

void friendly()
{
      using namespace std;
      cout << "Hello \n";
}


void shy(int audience_count)
{
      using namespace std;
      if (audience_count < 5)
           return;
      cout << "Goodbye \n";
}
```

2. Обязательно ли наличие оператора return в определении функции типа void?
3. Предположим, что мы удалили оператор return из определения функции initialize_screen в листинге 4.1. Как это отразится на выполнении программы? Будет ли она компилироваться и выполняться? А что произойдет, если удалить оператор return из определения функции show_results или функции celsius?
4. Определите функцию типа void с тремя аргументами типа int, выводящую на экран произведение этих трех значений. Включите данное определение в программу, запрашивающую у пользователя три числа, а затем вызовите функцию типа void.

[1] В стандарте C++ сказано, что оператор return 0 в функции main не является обязательным, однако многие компиляторы по-прежнему требуют его наличия.

5. Позволяет ли ваш компилятор определять в программе функции `int main()` и `void main()`? Какое предупреждение выводится, если функция определена как `int main()` и в ней отсутствует оператор `return 0`?
6. Является ли вызов функции типа `void` исполняемым оператором?

4.2. Передача параметров по ссылке

При вызове функции ее аргументам присваиваются значения формальных параметров. В главе 3 описывался механизм передачи параметров по значению. Однако это не единственный способ передачи параметров. В данном разделе мы рассмотрим другой механизм — *передачу параметров по ссылке.*

Первое знакомство с передачей параметров по ссылке

Использовавшийся нами до сих пор механизм передачи параметров по значению годится не для всех подзадач. Например, одной из наиболее распространенных подзадач является обработка одного или нескольких значений, введенных пользователем. Давайте вернемся к программе, приведенной в листинге 4.1. Ее задача включает четыре подзадачи: инициализация экрана, ввод температуры по Фаренгейту, вычисление соответствующего значения по Цельсию и вывод результатов. Три из этих подзадач реализованы посредством функций `initialize_screen`, `show_results` и `celsius`. Однако подзадача ввода данных реализована в составе функции `main` и содержит следующие строки кода:

```
cout << " I will convert a Fahrenheit temperature"
     << " to Celsius.\n"
     << "Enter a temperature in Fahrenheit: ";
cin >> f_temperature;
```

Ее также можно реализовать в виде вызова функции, и для этого мы воспользуемся механизмом передачи параметров по ссылке.

Функция, предназначенная для ввода данных, присваивает одной или нескольким переменным значения, введенные пользователем с клавиатуры. Таким образом в вызове функции задаются одна или несколько переменных, значения которых функция должна изменить. Если передавать параметры по значению, как мы поступали до сих пор, в качестве аргументов можно задать переменные, но вот их значения функция никак изменить не сможет. Ведь в этом случае в формальные параметры подставляются только значения переменных. А для нашей функции ввода нужно, чтобы в параметры подставлялись сами переменные. Именно так и работает механизм передачи параметров по ссылке. Если формальный параметр функции определен как передаваемый по ссылке, соответствующий ему аргумент должен быть переменной. При вызове функции эта переменная заменяет данный параметр, и все операции, выполняемые над параметром, производятся над переменной-аргументом. Поэтому, если функция изменяет значение параметра, значение переменной в вызывающем коде тоже изменяется.

Для того чтобы параметр *передавался по ссылке*, он должен быть соответствующим образом отмечен в заголовке функции и в ее объявлении, а именно: в конец имени типа этого параметра необходимо добавить символ амперсанда (&). Например, в приведенном ниже определении функции имеется один формальный параметр, передаваемый по ссылке:

```
void get_input(double& f_variable)
{
    cout << "I will convert a Fahrenheit temperature"
         << " to Celsius.\n"
         << "Enter a temperature in Fahrenheit: ";
    cin >> f_variable;
}
```

В программе, содержащей этот фрагмент программного кода, после следующего вызова функции get_input:

```
get_input(f_temperature);
```

переменной f_temperature будет присвоено значение, введенное пользователем с клавиатуры.

Используя данное определение функции, можно модифицировать программу, показанную в листинге 4.1, выделив из ее главной части подзадачу чтения. Но мы не будем переписывать эту программу, а рассмотрим новую (листинг 4.2).

Листинг 4.2. Передача параметров по ссылке

```
// Программа, демонстрирующая передачу параметров по ссылке.
#include <iostream>

void get_numbers(int& input1, int& input2);
// Функция считывает два целых числа, вводимых с клавиатуры.

void swap_values(int& variable1, int& variable2);
// Функция меняет местами значения переменных variable1 и variable2.

void show_results(int output1, int output2);
// Функция выводит значения переменных variable1 и variable2 в том порядке,
// в котором они были введены.

int main()
{
    int first_num, second_num;

    get_numbers(first_num, second_num);
    swap_values(first_num, second_num);
    show_results(first_num, second_num);
    return 0;
}

// Используем библиотеку классов iostream:
void get_numbers(int& input1, int& input2)
{
    using namespace std;
```

продолжение

Листинг 4.2 (*продолжение*)

```
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}
void swap_values(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

// Используем библиотеку классов iostream:
void show_results(int output1, int output2)
{
    using namespace std;
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Пример диалога

```
Enter two integers: 5 10
In reverse order the numbers are: 10 5
```

Задача этой программы проста: она считывает два числа и выводит их на экран в обратном порядке. Параметры функций `get_numbers` и `swap_values` передаются по ссылке. Считывание выполняется функцией

```
get_numbers(first_num, second_num);
```

В данной строке устанавливаются значения переменных `first_num` и `second_num`. Затем нижеприведенная функция меняет местами эти значения:

```
swap_values(first_num, second_num);
```

В следующих разделах описывается механизм передачи параметров по ссылке и рассматриваются функции, использующиеся в программе из листинга 4.2.

Передача параметров по ссылке

Чтобы определить формальный параметр функции как передаваемый по ссылке, нужно добавить в конец имени его типа данных символ &. Соответствующий аргумент в вызове функции должен быть переменной, а не константой или каким-либо выражением. При вызове функции переменная-аргумент (а не ее значение) подставляется в формальный параметр. В результате любые изменения значения формального параметра, осуществляемые в теле функции, фактически будут выполняться над переменной-аргументом. Более подробное описание механизма подстановки аргументов, передаваемых по ссылке, приведено далее в этой главе.

Пример определения параметров, передаваемых по ссылке, в объявлении функции:

```
void get_data(int& first_in, double& second_in)
```

Механизм передачи параметров по ссылке

В большинстве ситуаций механизм передачи параметров по ссылке работает так, словно формальный параметр в теле функции заменен именем переменной, заданной в качестве ее аргумента. Однако на самом деле этот механизм несколько сложнее, и в некоторых случаях важно точно знать, как он действует.

Напомним, что значение переменной программы хранится в отдельной ячейке памяти, адрес которой назначается компилятором. Например, при компиляции программы, приведенной в листинге 4.2, переменная first_num может получить адрес 1010, а переменная second_num – адрес 1012. Но для программиста это не имеет значения.

В качестве примера рассмотрим объявление функции get_numbers из программы листинга 4.2:

```
void get_numbers(int& input1, int& input2);
```

Передаваемые по ссылке формальные параметры input1 и input2 обозначают места подстановки реальных аргументов, заданных в вызове функции.

А вот сам вызов:

```
get_numbers(first_num, second_num);
```

При вызове функции в нее подставляются не имена переменных-аргументов, а их адреса. В нашем примере это

```
1010
1012
```

Первый адрес назначается первому формальному параметру, а второй – второму. Это можно представить следующим образом:

```
first_num ——► 1010 ——► input1
second_num——► 1012 ——► input2
```

Поэтому все заданные в теле функции операции над формальными параметрами на самом деле выполняются над переменными, расположенными в памяти по адресам, связанным с этими формальными параметрами. В данном случае инструкции в теле функции говорят, что первое значение, прочитанное с помощью оператора cin, должно быть сохранено в формальном параметре input1, а значит, в переменной, расположенной по адресу 1010 (то есть в переменной first_num). Второе значение должно быть сохранено в формальном параметре input2, а следовательно, в переменной, находящейся по адресу 1012 (в переменной second_num).

Подробно рассмотрим вызов функции с передачей аргументов по ссылке, используя пример функции из программы листинга 4.2.

1. Предположим, что компилятор назначил переменным first_num и second_num следующие адреса:

   ```
   first_num ——► 1010
   second_num——► 1012
   ```

(Мы не знаем, какие адреса назначены на самом деле, и результат не зависит от конкретных адресов. Наша цель — максимально точно и детально описать процесс.)

2. В программе (см. листинг 4.2) начинается выполнение следующей функции:

```
get_numbers(first_num, second_num);
```

3. Функции предписывается использовать адрес переменной first_num вместо формального параметра input1 и адрес переменной second_num вместо формального параметра input2. Результат получается таким же, как если бы определение функции было следующим (это не код C++, а скорее, псевдокод):

```
void get_numbers(int& переменная_по_адресу_1010>, int& переменная_по_адресу_1012)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> переменная_по_адресу_1010
        >> переменная_по_адресу_1012;
}
```

Поскольку переменные с адресами 1010 и 1012 — это first_num и second_num, результат получается таким же, как если бы функция была определена следующим образом:

```
void get_numbers(int& first_num, int& second_num)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> first_num
        >> second_num;
}
```

4. Выполняется тело функции, что эквивалентно работе такого кода:

```
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> first_num
        >> second_num;
}
```

5. При выполнении оператора cin переменным first_num и second_num присваиваются значения, введенные с клавиатуры. (В примере из листинга 4.2 переменной first_num присваивается значение 5, а переменной second_num — 10.)

6. По завершении вызова функции переменные first_num и second_num сохраняют значения, присвоенные им в теле функции с помощью оператора cin. (В примере, приведенном в листинге 4.2, переменная first_num содержит значение 5, а переменная second_num — значение 10.)

Может показаться, что мы описываем этот процесс слишком подробно. Если переменная first_num имеет адрес 1010, почему мы говорим «переменная, расположенная по адресу 1010», а не просто «first_num»? Дело в том, что имена аргументов и формальных параметров могут совпадать. Допустим, формальные параметры функции get_numbers названы input1 и input2. Предположим, что мы изменили программу, представленную в листинге 4.2, так, что указанной функции передаются

в переменные с такими же именами, input1 и input2, и эта программа выполняет нечто менее очевидное. В частности, она должна записать первое введенное число в переменную input2, а второе число — в переменную input1. Предположим также, что эти переменные, объявленные в главной части программы, имеют адреса 1014 и 1016. Вызов функции может быть таким:

```
int input1, input2;
get_numbers(input2, input1);   // Обратите внимание на порядок аргументов.
```

В данном случае, если просто сказать «input1», будет неясно, о чем идет речь — о переменной input1, объявленной в функции main, или об одноименном формальном параметре функции get_numbers. Если переменной input1, объявленной в функции main, назначен адрес 1014, выражение «переменная, расположенная по адресу 1014» однозначно. Давайте рассмотрим действие механизма передачи параметров по ссылке в этом случае.

Формальному параметру input1 соответствует аргумент input2, а формальному параметру input2 — аргумент input1. Это может показаться запутанным, но компьютер не увидит ничего странного, поскольку он не подставляет input1 в input2 или input2 в input1. Он имеет дело только с адресами в памяти и связывает переменную с адресом 1016 с формальным параметром input1, а переменную с адресом 1014 с формальным параметром input2.

Пример: функция swap_values

Функция swap_values, определение которой приведено в листинге 4.2, меняет местами значения двух переменных. Ее описание включает следующее объявление и сопутствующие комментарии:

```
void swap_values(int& variable1, int& variable2);
// Функция меняет местами значения переменных variable1 и variable2.
```

Для того чтобы понять, как эта функция работает, предположим, что в переменной first_num содержится значение 5, а в переменной second_num — значение 10, и рассмотрим следующий вызов:

```
swap_values(first_num, second_num);
```

После его выполнения переменная first_num получит значение 10, а переменная second_num — значение 5.

Определение функции swap_values в листинге 4.2 включает переменную temp. На первый взгляд кажется, что в ней нет необходимости и определение функции можно упростить:

```
void swap_values(int& variable1, int& variable2)
{
    variable1 = variable2;   // НЕВЕРНО
    variable2 = variable1;
}
```

Давайте разберемся, почему этот код не подходит для решения нашей задачи. При выполнении следующей строки кода:

```
swap_values(first_num, second_num);
```

переменные first_num и second_num подставляются вместо формальных параметров variable1 и variable2, так что вызов функции с приведенным неверным определением эквивалентен выполнению такого кода:

```
first_num = second_num;
second_num = first_num;
```

В результате получается не то, что нам нужно. Сначала переменной first_num присваивается значение переменной second_num, и это правильно. Но затем переменной second_num присваивается новое значение переменной first_num, которая теперь содержит то же значение, что и second_num. (Поэкспериментируйте с конкретными значениями переменных.) Поэтому, прежде чем присваивать переменной first_num значение переменной second_num, нужно сохранить ее исходное значение, для того чтобы оно не было потеряно. Для этого в программе из листинга 4.2 и используется локальная переменная temp. Вызов функции swap_values в данной программе эквивалентен выполнению следующего кода:

```
temp = first_num;
first_num = second_num;
second_num = temp;
```

Смешанные списки параметров

Способ передачи параметра функции определяется наличием символа & в конце имени его типа. Если амперсанд имеется, параметр передается по ссылке, в противном случае — по значению.

Параметры и аргументы

Для успешной работы с параметрами и аргументами необходимо освоить используемую терминологию. В частности, вам нужно знать следующее.

- *Формальные параметры* функции перечислены в ее объявлении и применяются в теле функции. Формальный параметр любого типа является лишь обозначением места подстановки аргумента или его значения при вызове функции.
- *Аргумент* (или *фактический параметр*) — это значение либо переменная, которые подставляются в формальный параметр. В вызове функции аргументы задаются в виде списка в скобках после ее имени. При вызове функции аргументы подставляются в соответствующие формальные параметры.

Термины *передача по значению* и *передача по ссылке* относятся к механизму передачи параметров. При передаче по значению в функцию передаются только значения аргументов, которыми инициализируются ее формальные параметры (являющиеся локальными переменными этой функции). При передаче по ссылке в качестве аргументов используются переменные. Эти переменные подставляются в тело функции вместо формальных параметров, так что все операции, относящиеся в определении функции к формальным параметрам, выполняются над переменными-аргументами.

В одной функции передачи вполне допустимо использовать параметры разных видов. Например, в следующем объявлении первый и третий параметры передаются по ссылке, а второй — по значению:

```
void good_stuff(int& par1, int par2, double& par3);
```

Передача параметров по ссылке возможна не только в функциях типа void, но и в функциях, возвращающих значение. Таким образом, в общем случае функция C++ может и возвращать, и изменять значение переменной-аргумента.

Совет программисту: как правильно выбрать способ передачи параметра

Программа, представленная в листинге 4.3, демонстрирует различие интерпретации компилятором формальных параметров, передаваемых по ссылке и по значению. В теле функции do_stuff двум ее параметрам — par1_value и par2_ref — присваиваются значения. Но поскольку эти параметры передаются по-разному, результаты такого присваивания различны.

Первый параметр, par1_value, передается по значению и является локальной переменной. При вызове функции

```
do_stuff(n1, n2);
```

локальная переменная par1_value инициализируется значением n1. Это означает, что локальная переменная par1_value инициализируется значением 1, а переменная n1 вообще игнорируется функцией. Как следует из этого примера, формальный параметр par1_value в теле функции инициализируется значением 111, и оно отображается на экране. Однако значение переменной-аргумента n1 не изменяется: программа выводит его на экран, и мы видим, что оно по-прежнему равно 1.

Второй параметр, par2_ref, передается по ссылке. При вызове функции переменная-аргумент n2 (а не просто ее значение) подставляется вместо формального параметра par2_ref. Поэтому выполнение следующего кода:

```
par2_ref = 222;
```

эквивалентно выполнению операции

```
n2 = 222;
```

Таким образом, в ходе выполнения тела функции значение переменной par2_ref (в начале работы программы инициализированной значением 2) изменяется и становится равным 222.

Ознакомившись с приведенным примером, можно сделать вывод: если нужно, чтобы функция изменяла значение переменной, соответствующий параметр необходимо передавать по ссылке, пометив его символом &. Во всех остальных случаях можно использовать механизм передачи параметров по значению.

Листинг 4.3. Сравнение механизмов передачи параметров

```
// Различие между передачей
// параметров по ссылке и по значению.
#include <iostream>

void do_stuff(int par1_value, int& par2_ref);
// Параметр par1_value передается по значению,
// а параметр par2_ref передается по ссылке.
int main()
```

продолжение ⇩

Листинг 4.3 (*продолжение*)

```
{
    using namespace std;
    int n1, n2;

    n1 = 1;
    n2 = 2;
    do_stuff(n1, n2);
    cout << "n1 after function call = " << n1 << endl;
    cout << "n2 after function call = " << n2 << endl;
    return 0;
}

void do_stuff(int par1_value, int& par2_ref)
{
    using namespace std;
    par1_value = 111;
    cout << "par1_value in function call = "
         << par1_value << endl;
    par2_ref = 222;
    cout << "par2_ref in function call = "
         << par2_ref << endl;
}
```

Пример диалога

```
par1_value in function call = 111
par2_ref in function call = 222
n1 after function call = 1
n2 after function call = 222
```

Ловушка: локальная переменная вместо параметра, передаваемого по ссылке

Когда нужно, чтобы функция изменяла значение переменной, соответствующий формальный параметр должен быть обозначен с помощью символа & (добавленного к имени его типа) как передаваемый по ссылке. Если случайно пропустить амперсанд, параметр будет передаваться по значению и после выполнения функция не будет изменять значение соответствующего аргумента. Формальный параметр, передаваемый по значению, является локальной переменной, поэтому изменение его значения, как и для любой другой локальной переменной, не отражается на программном коде вне тела функции.

В качестве примера рассмотрим программу, приведенную в листинге 4.4. Она идентична программе из листинга 4.2 с той разницей, что по ошибке в функции `swap_values` не введены символы амперсанда. Поэтому параметры `variable1` и `variable2` являются локальными переменными. Вместо них не подставляются переменные-аргументы `first_num` и `second_num` — параметры просто инициализируются значениями аргументов. Затем значения переменных `variable1` и `variable2` меняются местами, но переменные `first_num` и `second_num` сохраняют прежние значения. Как видите, отсутствие двух амперсандов делает программу абсолютно неправильной, хотя в остальном она идентична корректной программе, успешно компилируется и выполняется без сообщений об ошибках.

Листинг 4.4. Локальная переменная вместо параметра, переданного по ссылке

```
// Программа, демонстрирующая разницу между передачей параметров
// по ссылке и по значению.
#include <iostream>

void get_numbers(int& input1, int& input2);
// Функция считывает с клавиатуры два целых числа.

void swap_values(int variable1, int variable2);   // Отсутствует амперсанд.
// Функция меняет местами значения переменных variable1 и variable2.

void show_results(int output1, int output2);
// Функция выводит сначала значение переменной variable1, а затем variable2.

int main()
{
    int first_num, second_num;

    get_numbers(first_num, second_num);
    swap_values(first_num, second_num);
    show_results(first_num, second_num);
    return 0;
}

void swap_values(int variable1, int variable2)   // Отсутствует амперсанд.
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

// Используем библиотеку классов iostream:
void get_numbers(int& input1, int& input2)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> input1
        >> input2;
}

void swap_values(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

// Используем библиотеку классов iostream:
void show_results(int output1, int output2)
{
    using namespace std;
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Пример диалога

```
Enter two integers: 5 10
In reverse order the numbers are: 5 10
```

Упражнения для самопроверки

7. Что выводит следующая программа:

```
#include <iostream>
void figure_me_out(int& x, int y, int& z);
int main()
{
    using namespace std;
    int a, b, c;
    a = 10;
    b = 20;
    c = 30;
    figure_me_out(a, b, c);
    cout << a << " " << b << " " << c;
    return 0;
}

void figure_me_out(int& x, int y, int& z)
{
    using namespace std;
    cout << x << " " << y << " " << z << endl;
    x = 1;
    y = 2;
    z = 3;
    cout << x << " " << y << " " << z << endl;
}
```

8. Какой результат выведет программа, приведенная в листинге 4.2, если в определении функции `swap_values` символ `&` указать только для второго параметра?
9. Что выведет программа, приведенная в листинге 4.3, если изменить объявление функции `do_stuff` таким образом, чтобы параметр `par2_ref` передавался по значению:

```
void do_stuff(int par1_value, int& par2_ref);
```

10. Напишите определение функции `zero_both` типа `void` с двумя передаваемыми по ссылке параметрами типа `int`, которая устанавливает в `0` значения двух переменных.
11. Напишите определение функции `add_tax` типа `void`. У нее два формальных параметра: `tax_rate`, значение которого представляет налог с продажи в процентах, и `cost`, значение которого представляет стоимость товара без налога. Функция изменяет значение параметра `cost`, чтобы оно включало налог с продажи.
12. Может ли функция, возвращающая значение, иметь параметры, передаваемые по ссылке?

4.3. Использование процедурной абстракции

У меня такая плохая память,
что я часто забываю, как меня зовут.

Мигель де Сервантес Сааведра

Как вы помните, принцип процедурной абстракции гласит, что функции должны разрабатываться таким образом, чтобы их можно было использовать как черные ящики. Для выбора нужной функции программисту достаточно ознакомиться с ее объявлением и сопутствующим комментарием. Ему не требуется просматривать весь программный код, содержащийся в теле функции. В этом разделе рассматривается еще несколько тем, связанных с принципом процедурной абстракции.

Функции, вызывающие другие функции

Тело функции может содержать вызов другой функции. Собственно говоря, это не новость для вас: ведь все функции, с которыми вы до сих пор сталкивались, вызывались из главной части программы, то есть из функции `main`. Точно так же они могут быть вызваны из любой другой функции программы. Единственным условием является то, что вызову функции обязательно должно предшествовать ее объявление. Если вы решите создавать программы по образцу примеров этой книги, то есть все объявления функций выносить в начало программы, а определения располагать в конце, это условие будет соблюдаться автоматически. Хотя определение функции может содержать вызовы других функций, определений других функций в нем быть не может.

В листинге 4.5 представлена расширенная версия программы, приведенной в листинге 4.2. Программа из листинга 4.2 всегда меняет местами значения переменных `first_num` и `second_num`, тогда как программа из листинга 4.5 делает это только в определенных случаях в зависимости от условия

```
first_num <= second_num
```

Если оно выполняется, переменные остаются без изменений. Если же значение переменной `first_num` больше значения переменной `second_num`, вызывается функция `swap_values`, меняющая местами эти значения. Проверка значений переменных и принятие решения осуществляются в теле функции `order`, откуда и вызывается функция `swap_values`. В этом нет ничего особенного, поскольку в соответствии с принципом процедурной абстракции функция `swap_values` всегда выполняет определенную операцию (меняет местами значения двух переменных), и она не зависит от того, из какой части программы вызвана функция.

Предусловия и постусловия

В комментарии к объявлению функции полезно выделить два типа информации: *предусловие* и *постусловие*. Предусловие определяет, при каких условиях можно

вызывать данную функцию. Она не должна использоваться в других случаях, поскольку может работать неправильно. Постусловие описывает результаты вызова функции, то есть указывает, что будет происходить после выполнения функции при соблюдении предусловия. Если функция возвращает значение, в постусловии должно быть описано это значение. Если она изменяет значения некоторых переменных-аргументов, в постусловии должны быть описаны все эти изменения.

Например, комментарий к объявлению функции swap_values из программы листинга 4.5 можно было бы переписать так:

```
void swap_values(int& variable1, int& variable2);
// Предусловие: переменным variable1 и variable2
// присвоены значения.
// Постусловие: значения переменных variable1 и variable2
// поменялись местами.
```

Листинг 4.5. Вызов одной функции из другой

```
// Программа, демонстрирующая вызов одной функции из другой.
#include <iostream>

void get_input(int& input1, int& input2);
// Функция считывает с клавиатуры два целых числа.

void swap_values(int& variable1, int& variable2);
// Функция меняет местами значения переменных variable1 и variable2.

void order(int& n1, int& n2);
// Функция упорядочивает значения переменных n1 и n2 так,
// чтобы после вызова функции выполнялось условие n1 <= n2.

void give_results(int output1, int output2);
// Функция выводит значения переменных variable1 и variable2.
// Предполагается, что output1 <= output2.


int main()
{
    int first_num, second_num;

    get_input(first_num, second_num);
    order(first_num, second_num);
    give_results(first_num, second_num);
    return 0;
}
// Используем библиотеку классов iostream:
void get_input(int& input1, int& input2)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> input1 >> input2;
}

void swap_values(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
```

```
    variable1 = variable2;
    variable2 = temp;
}
// Эти определения функций могут следовать в любом порядке.
void order(int& n1, int& n2)
{
    if (n1 > n2)
        swap_values(n1, n2);
}

// Используется класс iostream:
void give_results(int output1, int output2)
{
    using namespace std;
    cout << "In reverse order the numbers are: "
         << output1 << " " << output2 << endl;
}
```

Пример диалога

```
Enter two integers: 10 5
In reverse order the numbers are: 5 10
```

А комментарий к объявлению функции `celsius`, приведенной в листинге 4.1, можно записать так:

```
double celsius(double fahrenheit);
// Предусловие: в переменной fahrenheit
// содержатся значения температуры по Фаренгейту.
// Постусловие: возвращает эквивалентные значения температуры
// в градусах Цельсия.
```

Если единственным постусловием является описание возвращаемого значения, программисты часто опускают слово *постусловие*. Распространенной и вполне приемлемой альтернативой приведенной выше форме комментария является следующая:

```
// Предусловие: в переменной fahrenheit
// содержатся значения температуры по Фаренгейту.
// Возвращает эквивалентные значения температуры
// в градусах Цельсия.
```

Вот еще один пример комментария с предусловием и постусловием:

```
void post_interest(double& balance, double rate);
// Предусловие: значение переменной balance определяет неотрицательный
// баланс на счету; значение переменной rate определяет величину
// начисления в процентах, например значение 5,
// соответствующее 5 %.
// Постусловие: значение переменной balance увеличено
// на значение переменной rate.
```

Как обычно, для использования функции `post_interest` не нужно просматривать ее определение — достаточно объявления и сопутствующего комментария.

Предусловия и постусловия — это не просто способ описания выполняемой функцией задачи. С их составления начинается создание самой функции. В ходе разработки программы сначала определяются задачи каждой из функций, а уже потом — способ ее реализации. В частности, объявления функций и сопутствующие

комментарии пишутся до начала работы над телом функции. Если впоследствии обнаружится, что по тем или иным причинам данную спецификацию реализовать невозможно, придется вернуться и пересмотреть задачи функции. Но их четкое определение в самом начале работы позволяет минимизировать как ошибки проектирования, так и время, затрачиваемое на разработку кода. Некоторые программисты не пользуются словами «предусловие» и «постусловие». Но информацию из предусловия и постусловия обязательно включают в комментарии.

Пример: цены в супермаркете

В этом примере решается очень простая задача, хотя может показаться, что она реализована слишком сложно.

Постановка задачи

Владельцы сети супермаркетов Quick-Shop поручили нам разработать программу, определяющую розничную цену заданного товара. В этих супермаркетах принята следующая схема ценообразования. Если предполагается, что товар будет распродан за неделю (7 дней) или менее, ему назначается скидка 5 %, а начиная с 8-го дня скидка на этот товар увеличивается до 10 %. Очень важно точно определить граничное значение, по достижении которого изменяется формула вычисления цены.

Как всегда, мы должны четко определить необходимые программе входные данные и генерируемую ею выходную информацию.

Входные данные

Входными данными являются оптовая цена товара и предполагаемое количество дней, в течение которых он будет продан.

Выходные данные

Выходные данные — это розничная цена товара.

Анализ задачи

Как и многие простые задачи, эта разбивается на три подзадачи:

- ввод данных;
- вычисление розничной цены товара;
- вывод результатов.

Реализуем подзадачи в виде трех функций. Ниже приведены их объявления и сопутствующие комментарии. Обратите внимание, что по ссылке передаются только те параметры, которые изменяются функциями; остальные определены как передаваемые по значению.

```
void get_input(double& cost, int& turnover);
// Предусловие: пользователь вводит корректные значения.
// Постусловие: переменной cost присвоено значение оптовой цены товара,
// а переменной turnover – ожидаемое количество дней,
// в течение которых товар будет продан.

double price(double cost, int turnover);
// Предусловие: в аргументе cost задана оптовая цена товара,
```

```
// а в аргументе turnover – ожидаемое число дней,
// в течение которых товар будет продан;
// возвращает значение розничной цены товара.

void give_output(double cost, int turnover, double price);
// Предусловие: в аргументе cost задана оптовая цена товара,
// а в аргументе turnover – ожидаемое количество дней,
// в течение которых товар будет продан; в аргументе price
// задана розничная цена товара.
// Постусловие: значения параметров cost, turnover и price
// выведены на экран.
```

Выполнив объявление функций, ничего не стоит написать главную часть программы:

```
int main()
{
    double wholesale_cost, retail_price;
    int shelf_time;

    introduction();
    get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    give_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}
```

И хотя мы еще не писали код тела функций и понятия не имеем, как он будет работать, уже сейчас можно полностью создать использующий их программный код. В этом и состоит принцип процедурной абстракции: функции интерпретируются как «черные ящики».

Разработка алгоритма

Функции get_input и give_output просты – они содержат по несколько операторов cin и cout. Алгоритм функции price можно представить в виде следующего псевдокода:

```
if turnover <= 7 дней
    return ( cost + 5 % от cost);
else
    return ( cost + 10 % от cost);
```

Написание программного кода

В нашей программе используются три постоянных значения: меньший процент (5 %), больший процент (10 %) и величина, определяющая границу между двумя диапазонами (7 дней), для которых используются разные скидки. Поскольку в том случае, если компания изменит схему ценообразования, эти значения придется менять прежде всего, мы объявим их в начале программы как глобальные именованные константы:

```
const double LOW_MARKUP = 0.05;     // 5 %
const double HIGH_MARKUP = 0.10;    // 10 %
const int THRESHOLD = 7;    // Константа HIGH_MARKUP используется в том случае,
                            // если товар НЕ предполагается продать
                            // за 7 или менее дней.
```

Алгоритм реализации тела функции price легко перевести на язык C++:

```
if (turnover <= THRESHOLD)
    return ( cost + (LOW_MARKUP * cost) );
else
    return ( cost + (HIGH_MARKUP * cost) );
```

Полный текст программы приведен в листинге 4.6.

Листинг 4.6. Программа формирования цен для супермаркетов

```
// Программа определяет розничную цену товара согласно схеме
// ценообразования, принятой в сети супермаркетов Quick-Shop.
#include <iostream>

const double LOW_MARKUP = 0.05;   // 5 %
const double HIGH_MARKUP = 0.10;  // 10 %
const int THRESHOLD = 7; // Константа HIGH_MARKUP используется в том случае,
                         // если товар НЕ предполагается продать
                         // за 7 или менее дней.

void introduction();
// Постусловие: на экран выводится описание программы.

void get_input(double& cost, int& turnover);
// Предусловие: пользователь вводит корректные значения.
// Постусловие: переменной cost присвоено значение оптовой цены товара,
// а переменной turnover – ожидаемое количество дней,
// в течение которых товар будет продан.

double price(double cost, int turnover);
// Предусловие: в аргументе cost задан значение оптовой цены товара,
// а в аргументе turnover – ожидаемое количество дней,
// в течение которых товар будет продан;
// возвращает значение розничной цены товара.

void give_output(double cost, int turnover, double price);
// Предусловие: в аргументе cost задано значение оптовой цены товара,
// а в аргументе turnover – ожидаемое количество дней,
// в течение которых товар будет продан; в аргументе price
// задано значение розничной цены товара.
// Постусловие: значения параметров cost, turnover и price
// выведены на экран.

int main()
{
    double wholesale_cost, retail_price;
    int shelf_time;

    introduction();
    get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    give_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}

// Используем библиотеку классов iostream:
void introduction()
```

```
{
    using namespace std;
        cout << "This program determines the retail price for\n"
             << "an item at a Quick-Shop supermarket store.\n";
}

// Используем библиотеку классов iostream:
void get_input(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}

// Используем библиотеку классов iostream:
void give_output(double cost, int turnover, double price)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Wholesale cost = $" << cost << endl
         << "Expected time until sold = "
         << turnover << " days" << endl
         << "Retail price = $" << price << endl;
}

// Используем именованные константы:
// LOW_MARKUP, HIGH_MARKUP и THRESHOLD:
double price(double cost, int turnover)
{
    if (turnover <= THRESHOLD)
        return (cost + (LOW_MARKUP * cost));
    else
        return (cost + (HIGH_MARKUP * cost));

}
```

Пример диалога

```
This program determines the retail price for
an item at a Quick-Shop supermarket store.
Enter the wholesale cost of item: $1.21
Enter the expected number of days until sold: 5
Wholesale cost = $1.21
Expected time until sold = 5 days
Retail price = $1.27
```

Тестирование программы

В ходе тестирования программы необходимо проверить все разновидности входных данных. В нашем случае они делятся на две категории: значения, для которых используется скидка 5 %, и значения, для которых используется скидка 10 %. Поэтому нам нужно протестировать как минимум по одному примеру из каждого диапазона.

Еще одна стратегия тестирования предполагает проверку *граничных значений*. К сожалению, эта концепция тоже не отличается определенностью. Входное (проверяемое) значение является граничным, если после его обработки изменяется поведение программы. Так, поведение нашей программы меняется после того, как значение второго из входных параметров станет равным 7. Для количества дней, которое меньше или равно 7, программа ведет себя иначе, чем в случае, когда число дней превышает 7. Поэтому нам нужно протестировать программу как минимум с одним набором входных значений, предполагающим, что товар будет оставаться на полке в точности 7 дней. Кроме того, обычно проверяют значения, прилегающие к граничному, поскольку при определении граничного значения программисты часто ошибаются. То есть нам нужно проверить программу для товара, который предположительно будет продан за 6 дней, а также для товара, который будет продан за 8 дней.

Упражнения для самопроверки

13. Может ли определение функции располагаться в теле другого определения функции?
14. Может ли определение функции содержать вызов другой функции?
15. Перепишите комментарий к объявлению функции `order` из программы, представленной в листинге 4.5, используя предусловие и постусловие.
16. Приведите предусловие и постусловие стандартной функции `sqrt`, вычисляющей квадратный корень числа.

4.4. Тестирование и отладка функций

Я увидел отвратительное существо —
жалкого созданного мною монстра.

Мэри Уолстонкрафт Шелли

Заглушки и отладочные программы

Каждая функция должна разрабатываться, кодироваться и тестироваться отдельно от остальной программы. В этом суть стратегии нисходящего проектирования. Когда функция рассматривается как отдельный модуль, одна большая задача разбивается на ряд мелких легкоуправляемых задач. Но как же тестировать функцию вне программы, для которой она предназначена? С этой целью пишется специальная программа, называемая *отладочной*. В листинге 4.7 приведена программа тестирования функции `get_input`, использовавшейся в программе из листинга 4.6. Такие отладочные программы являются временными средствами и должны содержать минимум кода — никаких затейливых подпрограмм ввода, никаких вычислений. Все, что нужно, — это получить значения (обычно от пользователя), которые следует передать функции в качестве аргументов, выполнить функцию и вывести результаты. Цикл (как в программе из листинга 4.7) позволит, не выходя из программы, повторить тестирование функции с разными аргументами.

Тестируя каждую функцию по отдельности, вы найдете большинство имеющихся в программе ошибок. Более того, узнаете, какие функции содержат эти ошибки. В то время как, обнаружив ошибку при тестировании программы целиком, трудно определить, где она находится.

Полностью протестированную функцию можно использовать в отладочной программе для тестирования другой функции. Главное, чтобы каждая функция тестировалась в программе, где все остальные функции уже протестированы. Только тогда не будет сомнений относительно источника очередной ошибки. Например, после полного тестирования функции get_input в отладочной программе из листинга 4.7 можно пользоваться ею для ввода, тестируя оставшиеся функции.

Иногда тестирование функции невозможно без другой функции, еще не написанной или не протестированной. В таких случаях пишут упрощенную версию необходимой функции, называемую *заглушкой*. Заглушка не обязательно должна выполнять все вычисления, входящие в задачи той функции, которую она заменяет: главное, чтобы она предоставляла необходимые для тестирования данные и была настолько простой, чтобы можно было не сомневаться в правильности ее работы. Так, программа листинга 4.8 предназначена для тестирования функции give_output из программы, представленной в листинге 4.6. Программа листинга 4.8 содержит функцию introduction, отдельно протестированную с помощью отладочной программы (которую мы здесь не рассматриваем). А вот функция price пока не протестирована, и поэтому вместо нее используется заглушка. Обратите внимание, что программу, приведенную в листинге 4.8, можно использовать даже в случае, если функция price еще не создана. Это позволяет тестировать общую схему основной программы до проработки деталей определений всех функций.

Программа с заглушками дает возможность протестировать каждую из функций непосредственно в основной программе, вместо того чтобы писать для них отдельные отладочные программы. При этом новые функции встраивается в уже проверенную программу и тестируются в ней. Поэтому тестирование основной программы с помощью заглушек обычно считается самым эффективным методом. Стратегия работы такова. Сначала посредством отладочной программы тестируется ряд базовых функций, таких как функции ввода и вывода. Затем начинается тестирование основной программы, в которой вместо всех еще не проверенных функций используются заглушки. Одна из заглушек заменяется функцией, эта функция тестируется, после чего подключается следующая функция — и так до тех пор, пока не будут по очереди подключены и протестированы все функции программы.

Листинг 4.7. Отладочная программа

```
// Отладочная программа для функции get_input.
#include <iostream>

void get_input(double& cost, int& turnover);
// Предусловие: пользователь вводит корректные значения.
// Постусловие: переменной cost присвоено зачение оптовой цены товара,
// а переменной turnover – ожидаемое количество дней,
// в течение которых товар будет продан.
```

продолжение ⇩

Листинг 4.7 (*продолжение*)

```
int main()
{
    using namespace std;
    double wholesale_cost;
    int shelf_time;
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        get_input(wholesale_cost, shelf_time);

        cout << "Wholesale cost is now $"
             << wholesale_cost << endl;
        cout << "Days until sold is now "
             << shelf_time << endl;

        cout << "Test again?"
             << "(Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}

// Используем библиотеку классов iostream:
void get_input(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}
```

Пример диалога

```
Enter the wholesale cost of item: $123.45
Enter the expected number of days until sold: 67
Wholesale cost is now $123.45
Days until sold is now 67
Test again? (Type y for yes or n for no): y

Enter the wholesale cost of item: $9.05
Enter the expected number of days until sold: 3
Wholesale cost is now $9.05
Days until sold is now 3
Test again? (Type y for yes or n for no): n
```

Основное правило тестирования функций

Каждая функция должна тестироваться в программе, где все остальные функции уже протестированы и отлажены.

Листинг 4.8. Программа с заглушкой

```
// Определяет розничную цену товара согласно схеме
// ценообразования, принятой в сети супермаркетов Quick-Shop.
#include <iostream>

void introduction();
// Постусловие: на экран выведено описание программы.

void get_input(double& cost, int& turnover);
// Предусловие: пользователь вводит корректные значения.
// Постусловие: переменной cost присвоено значение оптовой цены товара,
// а переменной turnover – ожидаемое количество дней,
// в течение которых товар будет продан.

double price(double cost, int turnover);
// Предусловие: в аргументе cost задано значение исходной цены товара,
// а в аргументе turnover – ожидаемое количество дней,
// в течение которых товар будет продан;
// возвращает значение розничной цены товара.

void give_output(double cost, int turnover, double price);
// Предусловие: в аргументе cost задано значение оптовой цены товара;
// в аргументе turnover – ожидаемое количество дней,
// в течение которых товар будет продан; в аргументе price
// задано значение розничной цены товара.
// Постусловие: значения параметров cost, turnover и price
// выведены на экран.

int main()
{
    double wholesale_cost, retail_price;
    int shelf_time;

    introduction();
    get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    give_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}

// Используем библиотеку классов iostream:
void introduction()   // Протестированная функция.
{
    using namespace std;
    cout << "This program determines the retail price for\n"
         << "an item at a Quick-Shop supermarket store.\n";
}

// Используем библиотеку классов iostream:
void get_input(double& cost, int& turnover)   // Протестированная функция.
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
```

продолжение ⇩

Листинг 4.8 (*продолжение*)

```
cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}

// Используем библиотеку классов iostream: тестируемая функция.
void give_output(double cost, int turnover, double price)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Wholesale cost = $" << cost << endl
         << "Expected time until sold = "
         << turnover << " days" << endl
         << "Retail price= $" << price << endl;
}

// Это только заглушка.
double price(double cost, int turnover)
{
    return 9.99; // Неправильно, но для тестирования допустимо.
}
```

Пример диалога

```
This program determines the retail price for
an item at a Quick-Shop supermarket store.
Enter the wholesale cost of item: $1.21
Enter the expected number of days until sold: 5
Wholesale cost = $1.21
Expected time until sold = 5 days
Retail price= $9.99
```

Упражнения для самопроверки

17. Сформулируйте основное правило тестирования функций. Чем хорош этот способ тестирования?
18. Что такое отладочная программа?
19. Напишите отладочную программу для функции introduction, которая приведена в листинге 4.8.
20. Напишите отладочную программу для функции add_tax из упражнения 11.
21. Что такое заглушка?
22. Создайте заглушку для функции, объявление которой приведено ниже.

```
double rain_prob(double pressure, double humidity, double temp);
// Предусловие: в параметре pressure задано значение давления в дюймах ртутного столба,
// в параметре humidity – значение относительной влажности в процентах,
// а в параметре temp – значение температруры в градусах по Фаренгейту.
// Возвращает число от 0 до 1, характеризующее вероятность выпадения осадков: 0 означает,
// что осадков наверняка не будет, а 1 – что осадки возможны с вероятностью 100 %.
```

Всю программу писать не нужно. Можем подсказать, заглушка будет очень короткой.

Резюме

- Все подзадачи программы можно реализовать как функции, возвращающие значение, либо как функции типа void.
- Формальный параметр — это обозначение места подстановки аргумента функции. Такая подстановка может выполняться одним из двух способов: по значению и по ссылке.
- При подстановке по значению значение аргумента подставляется в соответствующий формальный параметр. При подстановке по ссылке аргумент должен быть переменной, и эта переменная подставляется вместо соответствующего аргумента.
- Для того чтобы в определении функции обозначить параметр как передаваемый по ссылке, нужно добавить символ амперсанда в конец имени его типа.
- Функция не может изменить значение аргумента, переданного по значению. Если вы хотите, чтобы функция изменяла значение переменной, соответствующий параметр должен быть определен как передаваемый по ссылке.
- Удобным форматом комментария к объявлению функции является разделение информации на предусловие и постусловие. В предусловии указывается, в каких случаях допускается вызов данной функции. В постусловии описывается результат вызова функции, то есть условия, которые будут выполнены после вызова функции с соблюдением предусловия.
- Каждая функция должна быть протестирована в программе, содержащей полностью протестированные и отлаженные функции.
- Единственная задача отладочной программы — тестирование функции.
- Упрощенная версия функции называется заглушкой. Заглушка используется вместо определения еще не протестированной (и, возможно, даже не написанной) функции для тестирования остальной части программы.

Ответы к упражнениям для самопроверки

1. ```
   Hello
   Goodbye
   One more time:
   Hello
   End of program.
   ```
2. Нет, определение функции типа void может не содержать оператора return.
3. Удаление оператора return из определения функции initialize_screen, приведенного в листинге 4.1, никак не скажется на работе программы. Она будет успешно откомпилирована и сможет функционировать, как и раньше. Аналогично удаление оператора return из определения функции show_results никак не скажется на ее работе. А вот если исключить оператор return из определения
   ```

функции celsius, это будет серьезной ошибкой и программа не будет компилироваться. Дело в том, что функции initialize_screen и show_results относятся к типу void, а функция celsius возвращает результат.

4.
```
#include <iostream>

void product_out(int n1, int n2, int n3);

int main()
{
    using namespace std;
    int num1, num2, num3;
    cout << "Enter three integers: ";
    cin >> num1 >> num2 >> num3;
    product_out(num1, num2, num3);
    return 0;
}


void product_out(int n1, int n2, int n3)
{
    using namespace std;
    cout << "The product of three numbers "
         << n1 << ", " << n2 << ", and "
         << n3 << " равен " << (n1*n2*n3) << endl;
    return;
}
```

5. Ответы зависят от конкретной системы.

6. Вызов функции типа void, завершающийся точкой с запятой, является оператором. Вызов функции, возвращающей значение — выражением.

7.
```
10 20 30
1 2 3
1 20 3
```

8.
```
Enter two integers: 5 10
In reverse order the numbers are: 5 5
```

9.
```
par1_value in function call = 111
par2_ref in function call = 222
n1 after function call = 1
n2 after function call = 2
```

10.
```
void zero_both (int& n1, int& n2)
{
    n1 = 0;
    n2 = 0;
}
```

11.
```
void add_tax (double& tax_rate, double& cost)
{
     cost = cost + (tax_rate/100.0)*cost;
}
```

Деление на 100 выполняется для преобразования процента в дробь. Например, 10 % соответствуют 10/100, или 1/10 стоимости.

12. Да, функция, которая возвращает значение, может иметь параметры, передаваемые по ссылке.
13. Определение функции не может располагаться в теле другого определения функции.
14. Да, определение функции может содержать вызов другой функции.
15.
```
void order(int& n1, int& n2);
// Предусловие: в переменных n1 и n2 содержатся два числовых значения.
// Постусловие: значения переменных n1 и n2 упорядочены
// так, что n1 <= n2.
```
16.
```
double sqrt(double n);
// Предусловие: n>=0.
// Возвращает квадратный корень из n.
```
17. Одно из основных правил тестирования гласит, что каждая функция должна быть протестирована в программе, где все остальные функции протестированы и отлажены. Тогда, обнаружив ошибку, вы будете уверены, что она относится к тестируемой функции.
18. Отладочная программа – это программа, написанная исключительно для тестирования функции.
19.
```
#include <iostream>

void introduction();
// Постусловие: на экран выведено описание программы.

int main()
{
    using namespace std;
    introduction();
    cout << "End of test.\n";
    return 0;
}

// Используется класс iostream:
void introduction()
{
    using namespace std;
    cout << "This program determines the retail price for\n"
         << "an item at a Quick-Shop supermarket store.\n";
}
```
20.
```
// Отладочная программа для функции get_input.
#include <iostream>

void add_tax (double& tax_rate, double& cost);
// Предусловие: параметр tax_rate содержит значение налога с продажи в процентах,
// а параметр cost – значение стоимости товара без учета налога.
// Постусловие: параметр cost изменен, и теперь его значением является
// стоимость товара, включающая налоговую ставку.

int main()
{
    using namespace std;
    double cost, tax_rate;
```

```
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        cout << "Enter cost and tax rate:\n";
        cin >> cost >> tax_rate;
        add_tax(tax_rate, cost);

        cout << "After call to  add_tax \n"
             << "tax_rate is " << tax_rate << endl
             << "cost is " << cost << endl;

        cout << "Test again?"
             << "(Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}

void add_tax (double& tax_rate, double& cost)
{
     cost = cost + ( tax_rate/100.0 )*cost;
}
```

21. Заглушка — это упрощенная версия функции, используемая вместо нее для тестирования других функций.

22.
```
// ЭТО ТОЛЬКО ЗАГЛУШКА.
double rain_prob(double pressure, double humidity, double temp)
{
    return 0.25;   // Неправильно, но для тестирования годится.
}
```

Практические задания

1. Напишите программу, переводящую время из 24-часового в 12-часовой формат (например, 14:25 соответствует 2:25). Значение времени задается в виде двух целых чисел. В программе должны быть как минимум три функции: для ввода, для выполнения преобразований и для вывода. Значения AM и PM представьте как величины типа char: 'A' для AM (после полудня) и 'P' для PM (до полудня). Функция, выполняющая преобразование, будет иметь передаваемый по ссылке параметр типа char, в котором она вернет информацию о том, к какой половине суток относится результирующее время. (Она может иметь и другие параметры.) Включите в программу цикл, позволяющий повторять вычисления для других входных значений, пока пользователь не сообщит, что намерен выйти из программы.
2. Напишите функцию, вычисляющую среднеквадратическое отклонение четырех значений. Среднеквадратическое отклонение определяется как квадратный

корень среднего арифметического четырех значений $(s_i - a)^2$, где a – среднее арифметическое четырех значений, s_1, s_2, s_3 и s_4. Функция будет иметь шесть параметров и содержать вызовы еще двух функций. Включите функцию в отладочную программу, позволяющую тестировать ее снова и снова, пока пользователь не укажет, что тестирование окончено.

3. Напишите программу, сообщающую, монетами какого достоинства следует выдавать сдачу в размере от 1 до 99 центов. Например, если эта сумма составляет 86 центов, вывод функции будет примерно таким:

```
86 cents can be given as
3 quarter(s) 1 dime(s) and 1 penny(pennies)
```

Используйте только монеты номиналом 25 центов (quarter), 10 центов (dime) и 1 цент (penny).

```
void compute_coins(int coin_value, int& number, int& amount_left);
// Предусловие: 0 < coin_value < 100; 0 <= amount_left < 100.
// Постусловие: параметр number установлен равным максимальному
// количеству монет номиналом coin_value центов, которое можно
// получить из суммы в amount_left центов;
// значение параметра amount_left уменьшено на сумму,
// соответствующую количеству монет, то есть на значение выражения number*coin_value.
```

Для примера предположим, что значение переменной `amount_left` равно 86. После приведенного ниже вызова значением аргумента `number` будет 3, а значением аргумента `amount_left` – 11 (поскольку после вычитания из 86 центов суммы в 75 центов, выданной тремя монетами достоинством 25 центов, останется 11 центов):

```
compute_coins(25, number, amount_left);
```

Включите в программу цикл, позволяющий пользователю повторять эти вычисления желаемое количество раз.
Подсказка. для реализации функции используйте целочисленное деление и оператор %.

4. Напишите программу, считывающую длину в футах и дюймах и выводящую эту же длину в метрах и сантиметрах. Используйте три функции: для ввода, вычислений и вывода. Включите в программу цикл, который позволит пользователю повторять вычисления необходимое количество раз. (1 фут равен 0,3048 метра (или 12 дюймам), а 1 метр равен 100 сантиметрам.)
5. Напишите программу, аналогичную предыдущей, но преобразующую метры и сантиметры в футы и дюймы. Для выполнения этих подзадач используйте функции.
6. (Перед реализацией этого проекта нужно выполнить проекты 4 и 5.) Напишите программу, объединяющую функции из двух предыдущих проектов. Узнав у пользователя, какое преобразование он хочет выполнить: из футов и дюймов в метры и сантиметры или наоборот, программа производит заданное преобразование. Для выполнения первого преобразования пользователь вводит число 1, а для выполнения второго – 2. Программа считывает введенные данные и затем выполняет оператор `if...else`. В каждой его ветви вызывается одна из

двух функций. Определения этих функций очень похожи на программы из проектов 4 и 5. Они достаточно сложны и вызывают другие функции. Включите в программу цикл, позволяющий пользователю повторять вычисления желаемое количество раз.

7. Напишите программу, считывающую вес в фунтах и унциях и выводящую эквивалентный вес в килограммах и граммах. Используйте по меньшей мере три функции: для ввода, вычислений и вывода. Включите в программу цикл, позволяющий пользователю повторять вычисления желаемое количество раз. (В 1 килограмме 2,2046 фунта; 1 килограмм равен 1000 граммов, а в 1 фунте 16 унций.)
8. Напишите программу, аналогичную предыдущей, но преобразующую килограммы и граммы в фунты и унции. Для выполнения этих подзадач используйте функции.
9. (Перед реализацией этого проекта нужно выполнить два предыдущих.) Напишите программу, объединяющую функции из проектов 7 и 8. Выяснив у пользователя, какое преобразование он хочет выполнить: из фунтов и унций в килограммы и граммы или наоборот, программа производит заданное преобразование. Для выполнения первого преобразования пользователь вводит `1`, а для выполнения второго — `2`. Программа считывает введенные пользователем данные и затем выполняет оператор `if...else`. В каждой его ветви вызывается одна из двух функций. Определения этих функций очень похожи на программы из двух предшествующих проектов. Они достаточно сложны и вызывают другие функции. Включите в программу цикл, позволяющий пользователю повторять вычисления желаемое количество раз.
10. (Перед реализацией этого проекта нужно выполнить проекты 6 и 9.) Напишите программу, объединяющую функции из проектов 6 и 9. Программа спрашивает у пользователя, какое преобразование он хочет выполнить — длины или веса. Если выбрана длина, программа интересуется, какое преобразование нужно произвести — из футов и дюймов в метры и сантиметры или наоборот. Если выбран вес, программа задает аналогичный вопрос о фунтах, унциях, килограммах и граммах. Затем программа осуществляет заданное преобразование. Для выполнения первого преобразования пользователь вводит `1`, а для выполнения второго — `2`. Программа считывает данные, введенные пользователем, и затем выполняет оператор `if...else`. В каждой его ветви вызывается одна из двух функций. Определения этих функций очень похожи на программы из проектов 6 и 9. Заметьте, что в программе будут содержаться вложенные неявным образом операторы `if...else`. Внешний оператор `if...else` в зависимости от условия вызовет одну из двух функций, каждая из которых, в свою очередь, включает оператор `if...else`. Как и ранее, эти функции вызовутся в качестве «черных ящиков», а их детальная реализация будет выпол нена позднее. Если вы решите использовать четырехнаправленное ветвление, это будет, пожалуй, неудачным решением. Лучше воспользуйтесь двухнаправленными ветвлениями, хотя программа, обрабатывает четыре разных случая. Включите в программу цикл, позволяющий пользователю повторять вычисления желаемое количество раз.

11. Площадь произвольного треугольника можно вычислить по формуле:

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

где a, b и c — длины сторон, а s — половина периметра:

$$s = (a + b + c)/2$$

Напишите функцию типа `void` с пятью параметрами: посредством трех передаваемых по значению параметров задаются длины сторон, а с помощью передаваемых по ссылке параметров `area` и `perimeter` функция возвращает площадь и периметр треугольника. Она должна быть гибкой и надежной. Помните, что не все сочетания значений a, b и c образуют треугольник. Ваша функция должна возвращать правильные результаты для допустимых данных и идентифицировать недопустимые комбинации.

12. В холодную погоду метеорологи сообщают так называемый коэффициент резкости погоды, учитывающий скорость ветра и температуру. Посредством этого коэффициента оценивается охлаждающее воздействие ветра при заданной температуре. Его примерная формула такова:

$$W = 13{,}12 + 0{,}6125 * t - 11{,}37 * v^{0{,}16} + 0{,}3965 * t * v^{0{,}016}$$

где v — скорость ветра в м/с, t — температура в градусах по Цельсию ($t<=10$), W — коэффициент резкости погоды (в градусах по Цельсию).

Напишите функцию, возвращающую коэффициент резкости погоды. Программа должна проверять, соблюдаются ли ограничения по температуре. Сравните полученные результаты с коэффициентами, опубликованными в газетах.

Глава 5 Потоки ввода-вывода

— Озера наши и поток — ужель весь мир? — карась изрек.

Руперт Брук

Как лист несется потоком, впадающим в озеро или в море, так и вывод программы уносится потоком, и программа не знает, направляется он на экран или в файл.

Надпись в деканате факультета вычислительной техники

Ввод и вывод, то есть получение входных данных и выдача выходных — это операции, которые выполняются почти каждой программой. Входные данные могут быть получены от пользователя (введены с клавиатуры) или из файла. Выходные данные могут выводиться на экран или в файл. В этой главе рассказывается о том, как писать программы, способные считывать данные из файла и записывать их в другой файл.

Входные данные поступают в программу как поток, и в таком же виде выходные данные передаются на устройство вывода. Потоки можно рассматривать как первый пример *объектов*, с которыми вам предстоит познакомиться. Объект — это специфическая переменная, обладающая собственными функциями. В свое время язык C++ выделялся среди других языков программирования именно поддержкой объектов. Из этой главы вы узнаете, что такое потоки и как ими пользоваться для выполнения программного ввода-вывода. Попутно мы расскажем об основных концепциях использования объектов.

5.1. Потоки и основы файлового ввода-вывода

Честное слово, я и не подозревал, что вот уже более сорока лет говорю прозой.

Жан-Батист Мольер

Файл — вещь универсальная. В нем могут храниться как программа, так и данные, сгенерированные приложением или предназначенные для ввода. А чтобы

программы могли выводить данные на экран и в файлы одними и теми же универсальными способами, а также одинаковыми способами получать данные от пользователей и из файлов, в C++ реализована концепция потоков.

Поток — это последовательность символов (или других данных). Поток, передаваемый программе, называется *входным*, а поток, передаваемый программой, — *выходным*.

Вы уже не раз использовали потоки в программах, хотя, возможно, и не подозревали об этом. Знакомы вам и обозначения `cin` — имя входного потока (данные вводятся с клавиатуры) и `cout` — имя выходного потока (данные выводятся на экран). Эти потоки автоматически доступны в программе, содержащей директиву `#include` с заголовочным файлом `iostream`. Можно определить и другие передаваемые программе потоки — они используются точно так же, как потоки `cin` и `cout`.

Предположим, что в программе определен входной поток `in_stream`, поступающий из файла. (Как его определить, будет показано ниже.) Для помещения какого-либо значения из этого файла в переменную типа `int` достаточно поместить в программу следующий фрагмент кода:

```
int the_number;
in_stream >> the_number;
```

Если же в программе определен выходной поток, скажем, `out_stream`, передаваемый другому файлу, то в этот файл можно вывести и значение любой переменной. Например, следующий оператор выводит в файл передаваемое потоком `out_stream` значение переменной `the_number`, предваренное строкой `"the_number = "`:

```
out_stream << "the_number = " << the_number << endl;
```

Когда потоки соединены с нужными файлами, программа может точно так же выполнять файловый ввод-вывод, как ввод-вывод с использованием клавиатуры и экрана.

Почему для ввода-вывода используются файлы

Ввод с клавиатуры и вывод на экран, выполняемые рассмотренными выше программами, удобны для работы с временными данными — после завершения программы, введенная с клавиатуры или выведенная на экран информация пропадает. Файлы же обеспечивают возможность длительного хранения данных. Содержимое файла остается неизменным, пока его не модифицируют человек или программа. Если программа выводит данные в файл, по завершении ее работы он никуда не девается, а хранится на диске. А файл, содержащий входные данные, может применяться разными программами, и пользователь избавлен от необходимости каждый раз заново вводить одну и ту же информацию.

Используемые программами файлы ввода и вывода ничем не отличаются от тех, которые можно создавать и читать с помощью текстового редактора, например от файлов, содержащих исходный код программ. Это означает, что и файл с вводимыми данными, и файл вывода со сгенерированными программой данными можно

прочитать в любой момент, в отличие от информации, выводимой на экран, которая требует вашего постоянного внимания во время работы программы.

Кроме того, используя файлы, можно обрабатывать очень большие объемы информации. Если программе требуется получение огромного количества данных, проще вводить их из файла, а не с клавиатуры.

Файловый ввод-вывод

Когда программа получает входные данные из файла, говорят, что она их *считывает*; когда же программа выводит данные в файл, говорят, что она их *записывает*. Существуют разные способы чтения данных из файла, но мы при изложении материала будем применять только последовательное чтение — от начала к концу. Этот метод не позволяет программе возвращаться и повторно считывать уже обработанные данные. Как вы увидите далее, программа может повторить чтение файла, начав процесс с начала, но это будет именно «повтор», а не «возврат». Аналогичным образом, начиная с начала и продвигаясь к концу, программа выводит данные в файл. Она не может вернуться и изменить уже выведенные в файл данные, подобно тому, как нельзя исправить данные, ранее отображенные на экране. Для того чтобы иметь возможность выводить данные в файл или вводить их из файла, нужно соединить программу с этим файлом посредством потока.

В C++ роль потока выполняет переменная особого вида — *переменная-объект*. В следующем разделе мы поговорим об объектах подробнее, а пока посмотрим, как объекты-потоки используются для осуществления простейшего файлового ввода-вывода. Чтобы с помощью потока можно было ввести данные из файла или вывести их в него, нужно объявить этот поток в программе и соединить его с файлом.

Файл, с которым связан поток, можно представлять как значение потока. Поток может быть отсоединен от одного файла и соединен с другим — подобно тому, как можно удалить одно значение переменной и присвоить ей другое. Но для выполнения таких изменений используются специальные функции, предназначенные только для работы с потоками. Однако учтите, что хотя потоки и являются переменными, это не обычный вид переменных. Поэтому потоковую переменную нельзя использовать в операторе присваивания, как переменные типа `int` или `char`.

Потоки `cin` и `cout` являются предобъявленными, но поток для работы с файлом нужно объявлять, как любые другие переменные. Тип переменных (класс) для входных файловых потоков называется `ifstream` (от англ. input file stream), а для выходных — `ofstream` (от англ. output file stream). Приведенные ниже операторы объявляют переменную `in_stream` как входной файловый поток, а переменную `out_stream` — как выходной:

```
ifstream in_stream;
ofstream out_stream;
```

Классы `ifstream` и `ofstream` определены в заголовочном файле `fstream`, поэтому любая программа, в которой они применяются, должна содержать директиву `include` (обычно она располагается в начале программного кода):

```
#include <fstream>
```

Кроме того, в программе необходимо присутствие еще и директивы (также помещаемой в начало программы или в начало тела каждой функции), в которой используется класс ifstream или ofstream:

```
using namespace std;
```

Каждая потоковая переменная должна быть соединена с файлом, в частности, это касается объявленных выше переменных in_stream и out_stream. Для открытия файла и соединения его с потоком применяется функция open. Предположим, что входной поток in_stream нужно соединить с файлом infile.dat. Для этого перед чтением программой данных из файла в ней должен быть выполнен оператор

```
in_stream.open("infile.dat");
```

Синтаксис вызова функции open несколько необычен. Мы к нему еще вернемся, а пока обратите внимание на несколько интересных деталей. Во-первых, имени функции предшествуют имя переменной потока и точка. Во-вторых, имя файла заключено в кавычки. Оно передается функции open в качестве аргумента, и это то же самое имя, которое вы бы указали текстовому редактору при возникновении необходимости открыть в нем данный файл и внести изменения. Если входной файл находится в том же каталоге, что и программа, достаточно указать его имя, не уточняя путь. Если же файл хранится в другом каталоге, нужно указать его местонахождение.

Объявив потоковую переменную и соединив ее с файлом при помощи функции open, можно начинать ввод из файла с использованием оператора >>. Приведенный ниже код считывает два числа из файла, соединенного с потоком in_stream, и помещает их в переменные one_number и another_number.

```
int one_number, another_number;
in_stream >> one_number >> another_number;
```

Выходной поток открывается (то есть соединяется с файлом) точно так же, как и входной. Например, следующие две строки кода объявляют выходной поток out_stream и соединяют его с файлом outfile.dat:

```
ofstream out_stream;
out_stream.open("outfile.dat");
```

Если файла с заданным именем не существует, функция open, вызванная для потока класса ofstream, его создает. Если же этот файл имеется в наличии, функция удаляет его содержимое.

После соединения потока out_stream с файлом путем вызова функции open программа может выводить данные в файл с помощью оператора <<. Так, следующий оператор записывает в файл две строки и значения переменных one_number и another_number:

```
out_stream << "one_number = " << one_number
           << " another_number = " << another_number;
```

Обратите внимание, что при работе программы с файлом используется не одно, а два имени. Первое из них — обычное имя файла, применяемое операционной системой. Оно называется *внешним* и является настоящим именем файла. В нашем

примере такими именами являются infile.dat и outfile.dat.. Соглашения по применению внешних имен в разных системах различны, поэтому имена, используемые в наших примерах, для других систем могут не подойти. Хотя внешнее имя файла и является его настоящим именем, в программе оно употребляется только один раз в качестве аргумента функции open. Вторым именем файла в программе является имя соединенного с ним потока, поскольку на открытый файл всегда следует ссылаться именно по этому имени.

Демонстрационная программа, приведенная в листинге 5.1, считывает три числа из одного файла и записывает их сумму в другой.

Закончив работу с файлом, программа должна его закрыть, то есть отсоединить от него поток. Эта операция выполняется с помощью функции close. Как используется данная функция, показывают следующие строки из листинга 5.1:

```
in_stream.close();
out_stream.close();
```

Обратите внимание, что у функции close нет аргументов. Если работа программы завершается корректно, но она не закрывает файл, это делает операционная система. Однако рекомендуем вам взять за правило всегда закрывать файлы самостоятельно, и на то имеется как минимум две причины. Система закрывает файлы только если программа завершается правильно. В случае же аварийного ее завершения вследствие возникновения ошибки файл не будет закрыт и может оказаться поврежден. Если же программа сама закрывает файл по завершении работы, его порча менее вероятна. Вторым фактором, обуславливающим необходимость самостоятельного закрытия файла, является то обстоятельство, что, записав данные в файл, программа несколько позже может захотеть их оттуда прочитать. Для этого ей нужно закрыть файл, а затем подключить его к входному потоку с помощью функции open. (Файл можно открыть и для ввода и для вывода, но это делается несколько иным способом, который мы пока не описываем.)

Листинг 5.1. Простой файловый ввод-вывод

```
// Программа считывает три числа из файла infile.dat, находит их сумму
// и записывает ее в файл outfile.dat.
// (Дополненная версия этой программы приведена в листинге 5.2.)
#include <fstream>

int main()
{
    using namespace std;
    ifstream in_stream;
    ofstream out_stream;

    in_stream.open("infile.dat");
    out_stream.open("outfile.dat");

    int first, second, third;
    in_stream >> first >> second >> third;
    out_stream << "The sum of the first 3\n"
               << "numbers in infile.dat\n"
               << "is " << (first + second + third)
               << endl;
```

```
    in_stream.close();
    out_stream.close();
    return 0;
}
```

infile.dat
(Не изменяется программой)

```
1
2
3
4
```

outfile.dat
(После выполнения программы)

```
The sum of the first 3
numbers in infile.dat
is 6
```

Программа ничего не выводит на экран и не требует ввода данных с клавиатуры.

Введение в классы и объекты

Потоки in_stream и out_stream, о которых рассказывалось в предыдущем разделе, равно как и предопределенные потоки cin и cout, являются объектами. *Объект* — это переменная, которая помимо данных включает связанные с ней функции. Так, с каждым из потоков in_stream и out_stream связана функция open. Вот два примера ее вызова:

```
ifstream in_stream;
ofstream out_stream;
in_stream.open("infile.dat");
out_stream.open("outfile.dat");
```

Почему же для этих двух вызовов используется столь непривычный синтаксис? Функция open, связанная с объектом in_stream, и функция open, связанная с объектом out_stream, — две разные функции. Конечно, они похожи между собой, так как обе «открывают файлы», одна — для ввода, а другая — для вывода. Причиной того, что две функции получают одно и то же имя, является их назначение. Но поскольку это две разные функции, они могут несколько отличаться друг от друга. Встретив вызов функции с именем open, компилятор должен как-то определить, к какой из двух одноименных функций он относится. Вот для этой цели и предназначено имя объекта, указываемое перед именем функции и отделяемое от него точкой, в нашем случае — in_stream или out_stream. Функция, связанная с объектом, называется *функцией-членом*. Например, одна функция open является членом объекта out_stream, а другая — членом объекта in_stream.

Теперь вы знаете, что у объектов различных типов имеются разные функции-члены. Их имена могут совпадать, как в случае функций open, а могут различаться. Набор функций-членов объекта определяется его типом. Когда два объекта относятся к одному типу, у них могут быть разные значения, но одинаковые функции-члены. Если вы, предположим, объявите два объекта:

```
ifstream in_stream, in_stream2;
ofstream out_stream, out_stream2;
```

то здесь in_stream.open и in_stream2.open — это одна и та же функция, как и функции out_stream.open и out_stream2.open.

Тип, на основе которого создаются переменные-объекты, называется *классом*. Поскольку функции-члены объекта определяются его классом, то есть его типом, они называются *функциями-членами класса* или же *функциями-членами объекта*. Так, и у класса `ifstream`, и у класса `ofstream` имеется функция-член `open` (хотя во втором случае это уже другая функция). Но у класса `ofstream` к тому же есть функция-член `precision`, а у класса `ifstream` таковой нет. Вам уже приходилось пользоваться функцией `precision` потока `cout`, и позднее мы поговорим о ней подробнее.

При вызове функции-члена в программе всегда задается объект, имя которого обычно указывается перед именем функции и отделяется от него точкой, как в следующем примере:

```
in_stream.open("infile.dat");
```

Одна из причин, по которым в вызове функции указывается объект, заключается в том, что функция может оказывать на данный объект некоторое воздействие — в предыдущем примере вызов функции `open` соединяет файл infile.dat с потоком `in_stream`, поэтому ей необходимо знать имя потока.

В приведенном вызове точка является *оператором прямого доступа к члену класса*, а объект, указанный перед ней, — *вызывающим объектом*. В определенном смысле вызывающий объект представляет собой дополнительный аргумент функции — она может изменять его, как если бы он был аргументом. Однако его роль больше, чем у простого аргумента, поскольку он определяет значение имени функции, и без него компилятор не знал бы, какую именно из одноименных функций вызывает программа. Так, в приведенном примере функции `open` значение ее имени определяется объектом `in_stream`.

Помимо функций `open` у обоих классов, представляющих файловые потоки (`ifstream` и `ofstream`), имеются функции `close`. Обе они «закрывают файлы», но делают это различным образом, поскольку их объекты по-разному открывают и обрабатывают свои файлы. О других функциях-членах классов `ifstream` и `ofstream` будет рассказано далее в этой главе.

Классы и объекты

Объект — это переменная, с которой связаны функции. Такие функции называются функциями-членами. Класс — это тип, переменные которого являются объектами. Класс объекта (то есть его тип) определяет, какие функции-члены имеются у данного объекта.

Вызов функции-члена

Синтаксис

вызывающий_объект.имя_функции_члена(список_аргументов);

Примеры

```
in_stream.open("infile.dat");
out_stream.open("outfile.dat");
out_stream.precision(2);
```

Значение *имя_функции_члена* определяется классом (именем типа) *вызывающий_объект*.

Совет программисту: проверяйте, открыт ли файл

Вызов функции open не всегда приводит к открытию файла, и на то имеется ряд причин. Например, если вы пытаетесь открыть файл, чтобы получить из него данные, а файла с указанным именем не существует, вызов функции open не будет выполнен. Однако при этом сообщения об ошибке вы можете не получить, и программа продолжит работу непредусмотренным образом. Вот почему сразу после вызова функции open нужно проверить, открыт ли файл, и если это не так, выполнить соответствующие действия, скажем, завершить работу программы.

Для проверки результата выполнения операции над потоком используется функция-член fail, которая имеется и у класса ifstream, и у класса ofstream. Данная функция принимает два аргумента и возвращает значение типа bool. Ее вызов для потока in_stream производится следующим образом:

```
in_stream.fail()
```

Это логическое выражение, и его можно использовать в управляющих структурах, таких как цикл while и оператор if...else.

Вызов функции fail должен производиться после каждого вызова функции open; если вызов open не привел к открытию файла, функция fail возвращает значение true. Так, если неудачным оказывается следующий вызов функции open:

```
in_stream.open("infile.dat");
if (in_stream.fail())
{
    cout << "Input file opening failed.\n";
    exit(1);   // Завершает работу программы.
}
```

программа выводит сообщение об ошибке и завершается; если же файл открывается успешно, функция fail возвращает значение false и работа программы продолжается.

Поскольку fail является функцией-членом, она вызывается с использованием имени потока и оператора точка. Но учите, что вызов функции in_stream.fail относится только к вызову функции open входного потока in_stream, а не к другим вызовам функции open, выполняемым для иных потоковых объектов, даже если таковые относятся к тому же классу.

Приведенная выше функция exit не имеет ничего общего с классами и потоками, но часто используется в подобном контексте, вызывая немедленное завершение работы программы. Значение своего аргумента она возвращает операционной системе. Для того чтобы иметь возможность его применять, программа должна содержать директиву #include:

```
#include <cstdlib>
```

Кроме того, в программе необходима приведенная ниже директива, помещаемая в начало ее кода или в начало тела функции, где используется оператор exit:

```
using namespace std;
```

Функция exit — это стандартная функция с одним целочисленным аргументом. Если причиной ее вызова является возникновение ошибки, то ей передается значение 1, в противном случае — значение 0. В таких простых примерах, как наш, не важно, какое целочисленное значение задано в вызове данной функции, но для более сложных программ это может быть существенно.

В листинге 5.2 приведена та же программа, что и в листинге 5.1, дополненная проверкой результата вызова функции open. Файлы она обрабатывает по тому же принципу, что и программа из листинга 5.1. То есть, если файл infile.dat существует и содержит данные, приведенные в листинге 5.1, она создает файл outfile.dat, о содержимом которого также можно судить по указанному рисунку. Однако если что-то оказывается не так, программа из листинга 5.2 выводит на экран соответствующее сообщение и завершает свою работу. Например, если файла с именем infile.dat не существует, вызов функции open не производится и на экране появляется сообщение об ошибке.

Листинг 5.2. Файловый ввод-вывод с проверкой того, успешно ли открыт файл

```
// Программа считывает три числа из файла infile.dat, находит их сумму
// и записывает полученное значение в файл outfile.dat.
#include <fstream>
#include <iostream>
#include <cstdlib>

int main()
{
    using namespace std;
    ifstream in_stream;
    ofstream out_stream;

    in_stream.open("infile.dat");
    if (in_stream.fail())
    {
        cout << "Input file opening failed.\n";
        exit(1);
    }

    out_stream.open("outfile.dat");
    if (out_stream.fail())
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }

    int first, second, third;
    in_stream >> first >> second >> third;
    out_stream << "The sum of the first 3\n"
               << "numbers in infile.dat\n"
               << "is " << (first + second + third)
               << endl;

    in_stream.close();
```

```
    out_stream.close();

    return 0;
}
```

Вывод на экран, если файла infile.dat не существует

```
Input file opening failed.
```

Обратите внимание, что поскольку сообщение об ошибке должно отображаться на экране, для его вывода используется объект cout. А раз в программе применяется этот объект, мы добавили в нее директиву #include с указанием заголовочного файла iostream. (На самом деле в программе, содержащей директиву #include <fstream>, директива #include <iostream> не нужна, но и вреда от нее не будет, так как она сможет информировать читающего вашу программу о том, что вывод осуществляется не только в файл, но и на экран.)

Функция exit

Синтаксис

exit (*целочисленное_значение*);

В результате выполнения функции программа немедленно завершается. Аргумент *целочисленное_значение* может принимать любое значение, но в соответствии со стандартным соглашением 1 означает, что завершение работы программы обусловлено возникновением ошибки, а 0 используется во всех остальных случаях. Объявление функции exit содержится в заголовочном файле cstdlib. Поэтому в любой программе, где она используется, должны присутствовать следующие директивы:

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

Их не обязательно располагать друг за другом, а можно размещать там же, где и прочие директивы, уже хорошо вам знакомые.

Реализация файлового ввода-вывода

Мы уже говорили, что операторы >> и << одинаково работают как для потоков cin и cout, так и для потоков, соединенных с файлами. Однако стили программирования для файлового ввода-вывода и ввода-вывода с использованием экрана и клавиатуры различаются. Считывая данные с клавиатуры, нужно сначала запросить у пользователя информацию, а затем вывести ее на экран, вот так:

```
cout << "Enter the number: ";
cin >> the_number;
cout << "The number you entered is " << the_number;
```

Когда же программа вводит данные из файла, все эти операторы, использующиеся для взаимодействия с пользователем, не требуются. Единственное, что необходимо в таком случае — гарантия, что в файле находятся именно те данные, которые программа ожидает там найти. Она просто читает файл, предполагая, что нужные данные там имеются. Если in_file — потоковая переменная, соединенная

с соответствующим файлом, и вы хотите заменить приведенные выше операции ввода-вывода аналогичными операциями файлового ввода-вывода, эти три строки кода следует заменить единственной:

```
in_file >> the_number;
```

В программе может быть любое количество потоков, одновременно открытых для ввода и вывода. Поэтому одна и та же программа может запрашивать ввод данных с клавиатуры и из одного или нескольких файлов. Кроме того, возможна такая ситуация, когда все входные данные вводятся с клавиатуры, а вывод направляется как на экран, так и в файлы. Допускается любая комбинация входных и выходных потоков. В большинстве примеров этой книги используются объекты `cin` и `cout`, а ввод-вывод производится с использованием клавиатуры и экрана, но их легко модифицировать так, чтобы данные вводились из файла или выводились в файл.

Упражнения для самопроверки

1. Предположим, что вы пишете программу, в которой используются входной поток `fin`, соединенный с соответствующим файлом, и выходной поток `fout`, соединенный с файлом. Как вы объявите потоки `fin` и `fout`? Какие директивы `#include` (если таковые вообще будут необходимы) следует включить в файл программы?

2. Продолжая работу над программой из предыдущего упражнения, вы хотите, чтобы она вводила данные из файла stuff1.dat и выводила результаты в файл stuff2.dat. Какие операторы нужно использовать для соединения потока `fin` с файлом stuff1.dat, а потока `fout` с файлом stuff2.dat? Не забудьте ввести в программу функцию, проверяющую, успешно ли открываются файлы.

3. Допустим, что, продолжая писать программу из предыдущих двух упражнений, вы достигли точки, когда больше не нужно ни вводить данные из файла stuff1.dat, ни выводить их в файл stuff2.dat. Как можно закрыть эти файлы?

4. Предположим, вы хотите модифицировать приведенную в листинге 5.1 программу таким образом, чтобы она выводила данные не в файл outfile.dat, а на экран. (Входные данные по-прежнему должны поступать из файла infile.dat.) Какие изменения нужно внести в программу?

5. Какую директиву `#include` следует поместить в файл программы, если в ней используется функция `exit`?

6. Каково назначение аргумента функции `exit(1)`?

7. Предположим, что `bla` — объект, а `dobedo` — функция-член объекта `bla`, принимающая в качестве аргумента значения типа `int`. Как записать вызов функции `dobedo` объекта `bla` с аргументом `7`?

8. Какими свойствами файлов обладают обычные программные переменные? Чем от них отличаются потоковые переменные?

9. Назовите имена как минимум трех функций-членов объекта ifstream и приведите примеры использования каждой из них.
10. Программа считала половину строк файла. Что нужно сделать, чтобы она повторно прочитала первую строку?
11. Какие два имени есть у файла, с которым работает программа, и когда используется каждое из них?

Операторы файлового ввода-вывода

В следующем примере входные данные поступают из файла **infile.dat**, а выходные данные направляются в файл **outfile.dat**.

1. Поместите в код программы три директивы #include:

```
#include <fstream>    // Для файлового ввода-вывода
#include <iostream>   // Для объекта cout
#include <cstdlib>    // Для функции exit
```

2. Определите имя потока для входного файла (например, in_stream) и объявите для него переменную типа ifstream. Задайте имя потока для выходного файла (скажем, out_stream) и объявите для него переменную типа ofstream:

```
using namespace std;
ifstream in_stream;
ofstream out_stream;
```

3. С помощью функции-члена open соедините каждый поток с файлом, указав в качестве аргумента имя внешнего файла. Воспользовавшись функцией-членом fail, проверьте, успешно ли вызвана функция open:

```
in_stream.open("infile.dat");
if (in_stream.fail())
{
    cout << "Input file opening failed.\n";
    exit(1);
}

out_stream.open("outfile.dat");
if (out_stream.fail())
{
    cout << "Output file opening failed.\n";
    exit(1);
}
```

4. С помощью потока in_stream прочитайте входные данные из файла **infile.dat**:

```
in_stream >> some_variable >> some_other_variable;
```

5. Используя поток out_stream, запишите выходные данные в файл **outfile.dat**:

```
out_stream << "some_variable = " << some_variable << endl;
```

6. Закройте файлы, посредством функцией close:

```
in_stream.close();
out_stream.close();
```

Добавление данных в файл (факультативный материал)

Для вывода данных в файл в программе нужно организовать вызов функции-члена open, которая будет открывать файл и соединять с ним поток типа ofstream. До сих пор мы делали это, задавая функции open единственный аргумент — имя открываемого файла, поэтому тот всегда оказывался пустым (если файл с заданным именем уже существовал, его старое содержимое удалялось). Однако есть и другой способ открытия файла, при котором выводимые программой данные добавляются в конец уже имеющихся в нем данных.

Для добавления выходных данных программы в файл important.txt можно воспользоваться версией функции с двумя аргументами, как в следующем примере:

```
ofstream outStream;
outStream.open("important.txt", ios::app);
```

Если файла important.txt не существует, этот вызов его создаст, а если таковой уже имеется, то будет открыт с сохранением содержимого и весь вывод программы добавится в его конец. Такой способ работы с файлом демонстрирует программа, приведенная в листинге 5.3.

Второй аргумент функции open — ios::app — это особая константа, определенная в файле iostream. Для ее использования требуется следующая директива include:

```
#include <iostream>
```

Кроме того, программа должна содержать еще одну директиву, в которой применяется константа ios::app:

```
using namespace std;
```

Обычно эту директиву помещают либо в начало программного кода, либо в тело функции.

Листинг 5.3. Добавление данных в файл

```
// Программа добавляет данные в конец файла data.txt.
#include <fstream>
#include <iostream>

int main()
{
     using namespace std;

    cout << "Opening data.txt for appending.\n";
    ofstream fout;
    fout.open("data.txt", ios::app);
    if (fout.fail())
    {
        cout << "Input file opening failed.\n";
        exit(1);
    }

    fout << "5 6\n"
         << "7 8\n";
```

```
    fout.close();
    cout << "End of appending to file.\n";

   return 0;
}
```

data.txt
(Перед выполнением программы)

```
1 2
3 4
```

data.txt
(После выполнения программы)

```
1 2
3 4
5 6
7 8
```

Данные, выводимые на экран

```
Opening data.txt for appending.
End of appending to file.
```

Добавление данных в файл

Если данные нужно добавить в конец файла, сохранив его содержимое, то файл необходимо открыть следующим образом.

Синтаксис

выходной_поток.open(*имя_файла*, *ios::app*);

Пример

```
ofstream outStream;
outStream.open("important.txt", ios::app);
```

Использование имен файлов в качестве входных данных (факультативный материал)

До сих пор имена входных и выходных файлов задавались в программах литерально. Точнее, мы указывали их в качестве аргументов в вызовах функции open, как в следующем примере:

```
in_stream.open("infile.dat");
```

Однако это не всегда удобно. Так, программа, приведенная в листинге 5.2, считывает числа из файла infile.dat, суммирует их и выводит полученное значение в файл outfile.dat. Если вы захотите выполнить те же вычисления над числами из другого файла, например из файла infile2.dat, и записать их сумму в файл outfile2.dat, придется изменить имена файлов в двух вызовах функции open и перекомпилировать программу. Удобнее написать программу так, чтобы она запрашивала у пользователя имя файла, из которого следует взять входные данные, и имя файла, в который следует поместить выходные данные. Тогда ее можно будет без перекомпиляции выполнять сколько угодно раз с разными именами файлов.

Имя файла представлено строкой, а принцип обработки строк будет подробно рассмотрен в главе 10. Пока мы приведем лишь начальные сведения, достаточные

для того, чтобы вы могли ввести имя файла. Строка — это просто последовательность символов. Мы уже использовали строковые значения в операторах вывода, скажем таких:

```
cout << "This is a string.";
```

Кроме того, мы применяли строковые значения в качестве аргументов функции-члена open. Задаваемую литеральную строку, как в приведенном выше операторе cout, нужно заключать в двойные кавычки.

Для того чтобы программа могла запросить у пользователя имя файла, в ней должна быть объявлена переменная, подходящая для хранения строкового значения. Это делается следующим образом:

```
char file_name[16];
```

Данное объявление подобно объявлению переменной типа char с той разницей, что здесь за именем переменной в квадратных скобках следует целое число, определяющее максимальное количество символов, которое в ней может храниться. Такое число должно быть на единицу больше максимального количества символов строкового значения, содержащегося в переменной. Поэтому в нашем примере переменная file_name содержит строки длиной не более 15 символов. Имя file_name можно заменить любым другим идентификатором, поскольку это не ключевое слово, а значение 16 — любым положительным целым числом.

Строковые значения присваиваются переменным точно так же, как значения любых других типов. Рассмотрим в качестве примера такой фрагмент кода:

```
cout << "Enter the input file name (maximum of 15 characters):\n";
cin >> file_name;
cout << "OK, I will edit the file " << file_name << endl;
```

А вот пример диалога, происходящего при его выполнении:

```
Enter the input file name (maximum of 15 characters):
myfile.dat
OK, I will edit the file myfile.dat
```

После того как имя файла присвоено строковой переменной, такой как file_name, ее можно применять в программе как аргумент функции-члена open. Например, следующий код соединяет входной поток in_stream с файлом, имя которого является значением переменной file_name (и с помощью функции-члена fail проверяет, открыт ли файл):

```
ifstream in_stream;
in_stream.open(file_name);
if (in_stream.fail())
{
    cout << "Input file opening failed.\n";
    exit(1);
}
```

Заметьте: в том случае, когда строковая переменная используется в качестве аргумента функции-члена open, кавычки не применяются.

В листинге 5.4 приведена модифицированная версия программы, представленной в листинге 5.2. Она считывает данные из одного файла и выводит их в другой, но имена этих файлов задаются пользователем. Они вводятся с клавиатуры и присваиваются строковым переменным in_file_name и out_file_name, после чего эти переменные задаются в качестве аргументов в вызовах функции open. Обратите внимание на особенности объявления строковых переменных — за их именами следуют числа в квадратных скобках.

Строковые переменные — это не совсем обычные переменные, в частности, для изменения их значений нельзя использовать оператор присваивания.

Листинг 5.4. Ввод имени файла

```
// Программа считывает из заданного пользователем файла три числа,
// складывает их и записывает сумму в указанный пользователем файл.
#include <fstream>
#include <iostream>
#include <cstdlib>

int main()
{
    using namespace std;
    char in_file_name[16], out_file_name[16];
    ifstream in_stream;
    ofstream out_stream;

    cout << "I will sum three numbers taken from an input\n"
         << "file and write the sum to an output file.\n";
    cout << "Enter the input file name (maximum of 15 characters):\n";
    cin >> in_file_name;
    cout << "Enter the output file name (maximum of 15 characters):\n";
    cin >> out_file_name;
    cout << "I will read numbers from the file "
         << in_file_name << " and\n"
         << "place the sum in the file "
         << out_file_name << endl;

    in_stream.open(in_file_name);
    if (in_stream.fail())
    {
        cout << "Input file opening failed.\n";
        exit(1);
    }

    out_stream.open(out_file_name);
    if (out_stream.fail())
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }

    int first, second, third;
    in_stream >> first >> second >> third;
    out_stream << "The sum of the first 3\n"
               << "numbers in " << in_file_name << endl
               << "is " << (first + second + third)
               << endl;
```

продолжение

Листинг 5.4 (*продолжение*)

```
    in_stream.close();
    out_stream.close();

    cout << "End of program.\n";
    return 0;
}
```

numbers.dat
(Не изменяется программой)

```
1
2
3
4
```

outfile.dat
(После выполнения программы)

```
The sum of the first 3
numbers in infile.dat
is 6
```

Данные, выводимые на экран

```
I will sum three numbers taken from an input
file and write the sum to an output file.
Enter the input file name (maximum of 15 characters):
numbers.dat
Enter the output file name (maximum of 15 characters):
sum.dat
I will read numbers from the file numbers.dat and
place the sum in the file sum.dat
End of program.
```

5.2. Дополнительные средства выполнения потокового ввода-вывода

Вы увидите их на прекрасном листке с ручейком текста, струящимся меж полями.

Ричард Бринсли Шеридан

Форматирование выходных данных с помощью потоковых функций

Местоположение и форма представления выходных данных программы называется *форматом*. В C++ управлять форматом можно посредством команд, определяющих такие детали, как число пробелов между элементами и количество цифр после десятичной точки. Вы уже пользовались тремя операторами форматирования выходных данных, когда нужно было выводить денежные суммы обычным образом (не в экспоненциальном формате) с двумя цифрами после десятичной точки. Перед выводом денежной суммы в программу вставлялся код, который мы ранее называли «магической формулой»:

```
cout.setf(ios::fiXEd);
cout.setf(ios::showpoint);
cout.precision(2);
```

Теперь, когда вы освоили объектный синтаксис доступа к потокам, эту формулу можно объяснить и дополнить еще несколькими командами форматирования.

Прежде всего, все рассматриваемые здесь команды форматирования применимы к любому выходному потоку. Если программа выводит данные в файл, соединенный с выходным потоком out_stream, с помощью приведенных ниже команд можно обеспечить запись чисел с дробной частью в формате, обычном для записи денежных сумм.

```
out_stream.setf(ios::fiXEd);
out_stream.setf(ios::showpoint);
out_stream.precision(2);
```

Мы рассмотрим эти операторы в обратном порядке.

У каждого выходного потока имеется функция-член precision. Предположим, что в программе она вызывается для потока out_stream, как в приведенном выше примере. Тогда все последующие выводимые этим потоком числа отображаются либо с двумя значащими цифрами, либо с двумя цифрами после десятичной точки, что зависит от используемого вами компилятора. Приведем примеры выходных данных для компилятора, устанавливающего две значащие цифры:

```
23.   2.2e7   2.2   6.9e-1   0.00069
```

А вот примеры выходных данных для компилятора, устанавливающего две цифры после десятичной точки:

```
23.56   2.26e7   2.21   0.69   0.69e-4
```

В примерах настоящей книги компилятор устанавливает две цифры после десятичной точки.

Вызов функции precision относится только к указанному в нем потоку. Если в программе есть еще какой-то выходной поток, скажем с именем out_stream_two, вызов out_stream.precision относится только к выводу потока out_stream, но не к выводу потока out_stream_two. Однако функцию precision можно вызвать и для второго потока, задав для него другое количество цифр после десятичной точки, как в следующем примере:

```
out_stream_two.precision(3);
```

Остальные операторы форматирования в нашей магической формуле несколько сложнее функции precision. Приведем два вызова функции setf для потока out_stream:

```
out_stream.setf(ios::fixed);
out_stream.setf(ios::showpoint);
```

Имя данной функции является сокращением от англ. set flags — установить флаги. Флаг — это команда сделать что-либо одним из двух возможных способов. Когда флаг передается в качестве аргумента функции setf, он указывает компилятору записывать вывод в поток определенным образом — каким именно, зависит от флага.

В приведенном выше примере функция setf вызывается дважды, и эти два вызова устанавливают два флага: ios::fixed и ios::showpoint. Первый из них указывает, что поток должен выводить числа типа double *в формате с фиксированной запятой*,

то есть в том виде, в котором обычно записываются числа. Если вызовом функции setf этот флаг установлен, все числа с плавающей запятой (такие, как числа типа double) выводятся не в экспоненциальном, а в привычном для нас десятичном формате.

Флаг ios::showpoint указывает, что числа с плавающей запятой всегда должны содержать десятичный разделитель. Например, число 2.0 должно выводиться именно в таком виде, а не как 2. Иными словами, вывод будет включать десятичный разделитель, даже если после него стоят только нули. В таблице 5.1 приведен список наиболее распространенных флагов форматирования и их значений.

Таблица 5.1. Флаги форматирования для функции setf

Установленный флаг	Описание	Значение по умолчанию
ios::fixed	Числа с плавающей запятой не выводятся в экспоненциальном формате (его установка автоматически снимает флаг ios::scientific)	Не установлен
ios::scientific	Числа с плавающей запятой выводятся в экспоненциальном формате (его установка автоматически снимает флаг ios::fixed) Если ни один из флагов ios::fixed и ios::scientific не установлен, система сама решает, как вывести каждое из чисел	Не установлен
ios::showpoint	Для чисел с плавающей запятой всегда выводится десятичный разделитель и стоящие после него нули. Если же он не установлен, числа без дробной части могут выводиться без десятичного разделителя и следующих за ним нулей	Не установлен
ios::showpos	Перед положительными целочисленными значениями выводится знак плюс	Не установлен
ios::right	Если выполнен вызов функции-члена width, в котором задана ширина поля, каждый новый элемент будет выводиться с правого края поля заданной ширины. Для этого перед элементом будет указано необходимое количество пробелов. (Установка флага автоматически снимает флаг ios::left.)	Установлен
ios::left	Если выполнен вызов функции-члена width, в котором задана ширина поля, каждый новый элемент будет выводиться с левого края поля заданной ширины. Для этого после элемента будет указано необходимое количество пробелов. (Установка флага автоматически снимает флаг ios::right.)	Не установлен

Обратите внимание на флаг ios::showpos. Если он установлен для потока, как в следующем примере:

```
cout.setf(ios::showpos);
```

то перед всеми положительными числами выводится знак плюс.

Знак минус выводится перед отрицательными числами всегда, и никаких специальных флагов для этого устанавливать не нужно.

Еще одной распространенной функцией форматирования является функция `width`. В качестве примера рассмотрим ее вызов, выполненный для потока `cout`:

```
cout << "A";
cout.width(4);
cout << 2 << endl;
```

Этот код выводит на экран следующее:

```
A   2
```

Между буквой `A` и цифрой `2` вставлено три пробела. Функция `width` указывает потоку, сколько символов он должен выделить для выводимого элемента. В данном случае элемент (то есть число `2`) занимает всего один символ, а функция `width` определила для него четыре символа, поэтому слева от цифры `2` добавлено три пробела. Если выводимые данные требуют больше места, чем задано в аргументе функции `width`, они выводятся полностью независимо от значения, заданного в вызове функции.

Вызов функции `width` относится только к следующему за ним в программе выводимому элементу. Когда нужно вывести двенадцать чисел, каждое из которых должно занять по четыре символа, функцию `width` следует вызвать двенадцать раз. Если это неудобно, можно воспользоваться манипулятором `setw`, описанным далее.

Любой установленный флаг может быть снят с помощью функции `unsetf`. Например, после следующего вызова:

```
cout.unsetf(ios::showpos);
```

программа перестанет добавлять перед положительными числами знак плюс.

Почему некоторые аргументы называют флагами

Почему аргументы функции `setf`, например `ios::showpoint`, называют флагами, и что означает запись `ios::`?

Словом *флаг* обозначают что-то, что можно включать и выключать. Первоначально фразы с этим словом звучали примерно так: «Когда флаг поднят, сделайте то-то» или «Когда флаг опущен, сделайте то-то», но постепенно они трансформировались в современные «флаг установлен» и «флаг снят (или сброшен)». Когда с помощью функции `setf` устанавливается один из флагов, описанных в таблице 5.1, поток, для которого он установлен, начинает вести себя соответствующим образом.

Смысл обозначения `ios::` очень прост. Оно указывает, что следующий за ним идентификатор, такой как `showpoint` или `fixed`, относится к входному или выходному потоку — `ios`. Далее мы еще вернемся к обозначению `::` и поговорим о нем подробнее.

Манипуляторы

Манипуляторы — это специальные функции, которые в выражениях с потоковыми объектами записываются подобно обычным переменным, но в действительности

выполняют с потоком определенные действия. Данная функция связана с потоковым объектом и вызывает его функцию-член. Подобно выводимым элементам, манипуляторы помещаются после оператора вывода <<. Как и обычные функции, они могут иметь аргументы. Один из манипуляторов, endl, вам уже знаком, он вызывает переход на новую строку. Теперь мы расскажем еще о двух манипуляторах — setw и setprecision.

Назначение манипулятора setw и уже известной вам функции-члена width одинаково. Чтобы вызвать манипулятор setw, нужно записать его после оператора вывода <<, как если бы этот манипулятор направлялся в выходной поток. Например, код

```
cout << "Start" << setw(4) << 10
     << setw(4) << 20 setw(6) << 30;
```

выведет следующую строку:

```
Start  10  20    30
```

(Перед числом 10 выведено два пробела, перед числом 20 — еще два, а перед числом 30 — четыре пробела.)

Манипулятор setprecision подобен функции-члену precision (с которой вы уже знакомы). Однако его вызов записывается после оператора вывода <<, как вызов манипулятора setw. Так, следующий код выводит заданные числа с указанным в аргументе манипулятора setprecision количеством знаков после десятичной точки:

```
out_stream.setf(ios::fiXEd);
out_stream.setf(ios::showpoint);
cout << "$" << setprecision(2) << 10.3 << endl
     << "$" << 20.5 << endl;
```

Вот его вывод:

```
$10.30
$20.50
```

Когда количество знаков после десятичной точки устанавливается с помощью манипулятора setprecision, оно остается в силе до тех пор, пока не будет изменено с помощью вызова другого манипулятора setprecision или функции precision. (Напомним, что точно так же действует и функция precision.)

Для того чтобы в программе можно было пользоваться манипуляторами setw и setprecision, она должна содержать директивы:

```
#include <iomanip>
```

и

```
using namespace std;
```

Упражнения для самопроверки

12. Что выведет программа после выполнении таких строк ее кода (если в ней отсутствуют ошибки и она содержит необходимые директивы #include):

```
cout << "*";
cout.width(5);
cout << 123
```

```
    << "*" << 123 << "*" << endl;
cout << "*" << setw(5) << 123
    << "*" << 123 << "*" << endl;
```

13. Что выведет программа после выполнения следующих строк ее кода (если в ней отсутствуют ошибки и она содержит необходимые директивы #include):

```
cout << "*" << setw(5) << 123;
cout.setf(ios::left);
cout << "*" << setw(5) << 123;
cout.setf(ios::right);
cout << "*" << setw(5) << 123 << "*" << endl;
```

14. Что выведет программа после выполнения следующих строк ее кода (если в ней отсутствуют ошибки и она содержит необходимые директивы #include):

```
cout << "*" << setw(5) << 123 << "*"
    << 123 << "*" << endl;
cout.setf(ios::showpos);
cout << "*" << setw(5) << 123 << "*"
    << 123 << "*" << endl;
cout.unsetf(ios::showpos);
cout.setf(ios::left);
cout << "*" << setw(5) << 123 << "*"
    << setw(5) << 123 << "*" << endl;
```

15. Что программа выведет в файл stuff.dat после выполнения следующих строк ее кода (если в ней отсутствуют ошибки и она содержит необходимые директивы #include):

```
ofstream fout;
fout.open("stuff.dat");
fout << "*" << setw(5) << 123 << "*"
    << 123 << "*" << endl;
fout.setf(ios::showpos);
fout << "*" << setw(5) << 123 << "*"
    << 123 << "*" << endl;
fout.unsetf(ios::showpos);
fout.setf(ios::left);
fout << "*" << setw(5) << 123 << "*"
    << setw(5) << 123 << "*" << endl;
```

16. Что выведет программа после выполнения следующей строки ее кода (если в ней отсутствуют ошибки и она содержит необходимые директивы #include):

```
cout << "*" << setw(3) << 12345 << "*" << endl;
```

17. Для форматирования вывода могут использоваться такие передаваемые функции-члену потоковых объектов setf константы:

    а) ios::fixed;

    б) ios::scientific;

    в) ios::showpoint;

    г) ios::showpos;

    д) ios::right;

    е) ios::left.

    Каково действие каждой из них?

18. Следующий код считывает данные из файла infile.dat и выводит результаты в файл outfile.dat:

```
// Задача для самопроверки. Считывает данные из
// одного файла и выводит результаты в другой.
#include <fstream>
int main()
{
    using namespace std;
    ifstream instream;
    ofstream outstream;

    instream.open("infile.dat");
    outstream.open("outfile.dat");

    int first, second, third;
    in_stream >> first >> second >> third;
    out_stream << "The sum of the first 3\n"
               << "numbers in infile.dat\n"
               << "is " << (first + second + third)
               << endl;

    instream.close();
    outstream.close();

    return 0;
}
```

Какие изменения в него нужно внести, чтобы результаты выводились не в файл, а на экран? (Входные данные по-прежнему должны поступать из файла.)

Потоки в качестве аргументов функций

Поток может быть аргументом функции. Единственным ограничением является то, что он должен передаваться только по ссылке. Например, функция make_neat, приведенная в программе листинга 5.5, имеет два потоковых параметра: один типа ifstream для входного потока и один типа ofstream для выходного потока.

Листинг 5.5. Форматирование вывода

```
// Демонстрация применения команд форматирования вывода.
// Программа считывает три числа из файла rawdata.dat, а затем выводит их
// на экран и в файл neat.dat в соответствии с указанным форматом.
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>   // Необходимо для манипулятора setw.
using namespace std;
// Потоковые параметры должны передаваться по ссылке.
void make_neat(ifstream& messy_file, ofstream& neat_file,
               int number_after_decimalpoint, int field_width);
// Предусловие: потоки messy_file и neat_file соединены
// с файлами посредством функции open.
// Постусловие: числа из файла messy_file выведены
// на экран и в файл, соединенный с потоком neat_file.
// Числа выводятся по одному в строке в формате с
// фиксированной запятой (то есть не в научном формате);
```

```
// значение переменной number_after_decimalpoint определяет
// количество цифр после десятичной точки;
// перед каждым числом выводится знак плюс или минус, и ширина
// поля каждого числа равна значению переменной field_width.
// (Данная функция не закрывает файл.)

int main()
{
    ifstream fin;
    ofstream fout;

    fin.open("rawdata.dat");
    if (fin.fail())
    {
        cout << "Input file opening failed.\n";
        exit(1);
    }

    fout.open("neat.dat");
    if (fout.fail())
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }


    make_neat(fin, fout, 5, 12);

    fin.close();
    fout.close();

    cout << "End of program.\n";
    return 0;
}

// Используем библиотеки классов iostream, fstream и iomanip.
void make_neat(ifstream& messy_file, ofstream& neat_file,
               int number_after_decimalpoint, int field_width)
{
    neat_file.setf(ios::fixed);       // Не в научном формате.
    neat_file.setf(ios::showpoint);   // Выводить десятичную точку.
    neat_file.setf(ios::showpos);     // Выводить знак плюс.
    neat_file.precision(number_after_decimalpoint);
    cout.setf(ios::fiXEd);
    cout.setf(ios::showpoint);
    cout.setf(ios::showpos);
    cout.precision(number_after_decimalpoint);

    double next;
    while (messy_file >> next)  // Удовлетворяется, если очередное число прочитано из файла.
    {
        cout << setw(field_width) << next << endl;
        neat_file << setw(field_width) << next << endl;
    }
}
```

rawdata.dat
(Не изменяется программой)

```
 10.37      -9.89897
 2.313    -8.950   15.0

   7.33333    92.8765
-1.237568432e2
```

neat.dat
(После выполнения программы)

```
  +10.37000
   -9.89897
   +2.31300
   -8.95000
  +15.00000
   +7.33333
  +92.87650
 -123.75684
```

Данные, выводимые на экран

```
  +10.37000
   -9.89897
   +2.31300
   -8.95000
  +15.00000
   +7.33333
  +92.87650
 -123.75684
End of program.
```

Совет программисту: проверка на конец файла

И это все, и больше ничего.

Этель Барримор

Когда программа считывает данные из файла, часто их количество ей неизвестно — она просто должна прочитать все содержимое файла. Например, если в файле содержатся числа, программа может прочитать их все и вычислить среднее арифметическое, сколько бы чисел там не оказалось. Для этого ей нужно считывать их по одному до тех пор, пока в файле не останется ни одного не прочитанного числа. Предположим, что с файлом, содержащим набор чисел, соединен поток `in_stream`. Тогда алгоритм вычисления их среднего арифметического может быть таким:

```
double next, sum = 0;
int count = 0;
while (В файле еще есть непрочитанные числа.)
{
    in_stream >> next;
    sum = sum + next;
    count++;
}
Среднее арифметическое = sum/count.
```

Данный алгоритм очень близок к коду C++, осталось только выразить на этом языке следующее условие:

```
(В файле еще есть непрочитанные числа.)
```

Хотя это и выглядит странно, его можно представить так:

```
(in_stream >> next)
```

В результате приведенный выше алгоритм можно переписать следующим образом:

```
double next, sum = 0;
int count = 0;
while (in_stream >> next)
{
    sum = sum + next;
    count++;
}
Среднее арифметическое = sum/count.
```

Обратите внимание, что тело цикла не такое, как в псевдокоде. Поскольку оператор `in_stream >> next` теперь используется в качестве логического выражения, он удален из тела цикла.

В этом необычном цикле выражение `in_stream >> next` служит одновременно и для ввода очередного числа из потока `in_stream`, и для проверки условия цикла `while`. Таким образом, оно является и оператором, выполняющим некоторое действие, и логическим выражением. Как оператор оно вводит число из входного потока, а как логическое выражение возвращает значение `true` или `false`. Если в потоке имеется еще оно число, оно считывается и логическое выражение оказывается истинным, в результате чего тело цикла выполняется еще один раз. Если же чисел больше не осталось, ничего не вводится и логическое выражение оказывается ложным, в результате чего цикл завершается. В этом примере переменная `next` имеет тип `double`, но такой метод проверки на конец файла одинаково действует и для других типов данных, таких как `int` или `char`.

О пространствах имен

До сих пор мы с вами старались использовать для определений функций локальные директивы `using`. Это удобно, но как быть, если тип параметра функции требует указания пространства имен? Так, в программе листинга 5.5 использовались имена потоковых типов, относящиеся к пространству имен `std`, поэтому директива `using` в нем была задана вне тела функции. В подобных случаях директивы `using` удобнее всего размещать в начале программного кода после директив `#include`, как в листинге 5.5.

Помещение единственной директивы `using` в начало программного кода решает проблему, но многие специалисты не считают это решение наилучшим, так как оно не позволяет использовать два пространства имен с одинаковыми именами. Пока оно нам подходит, поскольку мы пользуемся только пространством имен `std`[1]. В главе 9 будет описан еще один способ решения проблемы.

[1] На самом деле мы пользуемся двумя пространствами имен — `std` и так называемым глобальным пространством имен, состоящим из всех имен, не входящих в другие пространства. Однако эта техническая подробность для нас сейчас не важна.

Многие программистов предпочитают помещать директивы using в начало кода программы. Рассмотрим в качестве примера директиву

```
using namespace std;
```

В этой книге в большинстве программ она размещена не в начале кода программы, а в начале кода каждой функции, где используется пространство имен std (сразу за открывающей фигурной скобкой), как в листинге 5.3. Еще лучший пример приведен в листинге 4.11. Все программы, рассмотренные нами до сих пор, и почти все приведенные далее будут работать одинаково независимо от того, где располагается директива using namespace std — в начале всего кода или в начале кода каждой функции. Директиву using для пространства имен std в большинстве случаев можно совершенно спокойно помещать в начало программного кода, но для других пространств имен это подходит не всегда.

Мы настоятельно рекомендуем располагать директивы using в определении функции (или других маленьких блоков кода). Пока вы используете только одно пространство имен, но учтите, что выработанная уже сейчас привычка облегчит вам работу в будущем, когда придется создавать более сложные программы. А пока мы и сами иногда отступаем от этого полезного правила, когда следование ему становится слишком обременительным или мешает рассмотрению других вопросов программирования. Если вы изучаете данный курс с преподавателем, поступайте согласно его требованиям, если нет — делайте, как вам удобнее.

Пример: форматирование файла

Программа, приведенная в листинге 5.5, вводит данные из файла rawdata.dat и выводит их на экран и в файл neat.dat в отформатированном виде. Числа выводятся по одному в строке, и длина каждого из них составляет двенадцать символов; для этого перед каждым числом добавляется соответствующее количество пробелов. Программа представляет их не в научном (экспоненциальном), а в стандартном десятичном формате. Все числа выводится с пятью цифрами после десятичного разделителя и со знаком плюс или минус. На экран выводятся в точности те же данные, что и в файл, плюс одна дополнительная строка, сообщающая об окончании работы программы. В программе используется функция make_neat, получающая в качестве параметров входные и выходные потоки, соединенные с соответствующими файлами.

Упражнения для самопроверки

19. Что выведет программа (если она написана без ошибок и содержит необходимые директивы #include) при выполнении содержащихся в ней следующих строк кода:

```
ifstream ins;
ins.open("list.dat");
int count = 0, next;
while (ins >> next)
{
    count++;
    cout << next << endl;
}
```

```
ins.close();
cout << count;
```

при условии, что в файле list.dat находятся такие данные: 1, 2, 3?

20. Напишите определение функции типа void с именем to_screen. У нее один формальный параметр file_stream типа ifstream. Предусловие и постусловие функции таковы:

```
// Предусловие: поток file_stream соединен с файлом посредством
// вызова функции-члена open. В файле содержится набор целых
// чисел (и ничего больше).
// Постусловие: числа из файла, соединенного с потоком
// file_stream, выведены на экран каждое в отдельной
// строке. (Эта функция не закрывает файл.)
```

21. (Упражнение для изучивших факультативный раздел «Использование имен файлов в качестве входных данных».) Предположим, что следующий фрагмент кода, включающий объявление строковой переменной и оператор ввода, входит в состав программы, не содержащей ошибок:

```
#include <iostream>
using namespace std;
// ...
char name[21];
cout >> name;
```

Какова максимальная длина имени файла, которое можно присвоить строковой переменной name?

5.3. Символьный ввод-вывод

Полоний: Что вы читаете, мой господин?
Гамлет: Слова, слова, слова.

Уильям Шекспир

Любые данные вводятся и выводятся как символьные. Например, если выходными данными программы является число 10, то на самом деле выведены два символа — '1' и '0'. Подобным же образом, если пользователь хочет ввести число 10, он вводит символ '1', а за ним символ '0'. А уж то, каким образом компьютер интерпретирует эти символы: как число 10 или как пару символов, зависит от конкретной программы. И как бы она не была написана, аппаратное обеспечение компьютера всегда считывает символы '1' и '0', а не число 10. Это преобразование символов в числа обычно выполняется автоматически, и вам не приходится задумываться о том, как это происходит. Но иногда автоматическое преобразование может оказаться не таким уж полезным. Поэтому в C++ предусмотрены низкоуровневые средства для ввода и вывода символьных данных, не выполняющие такого преобразования. С их помощью можно обойти автоматическое преобразование и выполнить вывод в точности так, как требуется конкретной программе. Можно даже написать функции ввода и вывода для работы с числами, записываемыми римскими цифрами.

Функции-члены get и put

Функция get позволяет программе считать один вводимый символ и сохранить его в переменной типа char. Такую функцию имеет каждый входной поток, независимо от того, соединен он с файлом или же данные вводятся с клавиатуры. Мы рассмотрим функцию-член get потока cin, но все сказанное в этом разделе в равной степени относится и к одноименной функции входного файлового потока.

До сих пор мы использовали объект cin с оператором ввода >>, считывая входные данные в переменные программы. При этом многие операции, связанные с чтением и интерпретацией вводимых пользователем значений (такие, как пропуск пробелов), выполнялись автоматически. В отличие от оператора >>, функция get ничего не делает автоматически. Поэтому если вы захотите пропустить введенные пользователем ведущие пробелы, придется написать программный код, анализирующий входные данные и удаляющий из них требуемые пробелы.

Функция-член get принимает один аргумент, который должен быть переменной типа char. Его значением является символ, прочитанный из входного потока. Например приведенный ниже код считывает следующий введенный с клавиатуры символ и сохраняет его в переменной next_symbol.

```
char next_symbol;
cin.get(next_symbol);
```

Важно, что программа может таким образом прочитать любой символ. Если следующим символом является пробел, приведенный код не пропускает его, а прочитывает и присваивает переменной next_symbol значение, равное символу пробела. Когда следующим вводится символ перевода строки '\n', означающий, что программа достигла конца вводимой строки, вызов cin.get присваивает переменной next_symbol значение '\n'.

Хотя символ перевода строки записывается как пара символов '\n', в C++ они интерпретируются как один. В качестве примера предположим, что программа содержит такой код:

```
char c1, c2, c3;
cin.get(c1);
cin.get(c2);
cin.get(c3);
```

и допустим, что пользователь ввел следующие две строки, предназначенные для прочтения этим кодом:

```
AB
CD
```

То есть он ввел символы AB, затем нажал клавишу Enter, ввел символы CD и снова нажал клавишу Enter. В результате, как нетрудно предположить, переменной c1 присвоено значение 'A', а переменой c2 значение 'B'. Пока в этом нет для нас ничего нового. Однако когда очередь доходит до переменной c3, ситуация меняется. Значение, получаемое этой переменной при вводе данных с помощью функции-члена get, отличается от того, которое присвоил бы ей оператор >>. При выполнении

приведенного выше кода переменная c3 получает значение '\n', то есть ей присваивается символ перевода строки, а не символ 'C'.

Функция get может понадобиться программе для того, чтобы определить, где во входных данных находятся концы строк. Например, следующий цикл считывает вводимую строку и останавливается по достижении ее конца (символа '\n'). Следующая вводимая строка будет считываться с начала, то есть с первого символа. В примере мы выполняем вывод введенных данных на экран, но в общем случае программа может делать с такими данными все, что потребуется.

```
cout << "Enter a line of input and I will echo it:\n"
char symbol;
do
{
    cin.get(symbol);
    cout << symbol;
} while (symbol != '\n');
cout << "That's all for this demonstration.";
```

Этот цикл считывает любую введенную строку и в точности повторяет ее на экране со всеми пробелами, символами табуляции и перевода строки. Ниже показаны данные, выводимые на экран при выполнении приведенного цикла:

```
Enter a line of input and I will echo it:
Do Be Do 1 2    34
Do Be Do 1 2    34
That's all for this demonstration.
```

Обратите внимание, что символ перевода строки '\n' и вводится, и выводится программой. Поэтому текст That's all for this demonstration. выводится с новой строки.

Еще одна функция-член объектов-потоков, называемая put, аналогична функции get с той разницей, что используется для вывода, а не для ввода. Она позволяет программе вывести один символ. Эта функция принимает один аргумент, который должен иметь тип char, например константу или переменную. При вызове функции put значение аргумента выводится в поток. Так, следующий оператор выводит на экран букву a:

```
cout.put('a');
```

В отличие от функции cin.get, функция cout.put не предоставляет программисту никаких дополнительных возможностей по сравнению с оператором вывода <<, но мы рассмотрим ее для полноты изложения.

При использовании в программе функций cin.get и cout.put в нее должна быть включена директива

```
#include <iostream>
```

необходимая для любых операций с объектами cin и cout.

Точно так же, если программа получает данные из файла с помощью функции get или выводит данные в файл посредством функции put, в ней должна содержаться директива

```
#include <fstream>
```

'\n' и "\n"

В некоторых случаях может показаться, что выражения '\n' и "\n" в программе эквивалентны. Например, в операторе cout с одинаковым эффектом можно использовать любое из них, но они отнюдь не всегда взаимозаменяемы. Выражение '\n' представляет значение типа char и может храниться в переменной этого типа. А выражение "\n" представляет строку, состоящую из единственного символа; оно относится к другому типу данных и не может храниться в переменной типа char.

В дополнение к любой из этих двух директив #include программа должна содержать директиву

```
using namespace std;
```

Функция-член get

Каждый входной поток имеет функцию-член get, с помощью которой можно считать один символ вводимых данных. В отличие от оператора >> функция get считывает любой следующий символ независимо от того, что это за символ. В частности, она позволяет считывать пробел, символ табуляции или перевода строки. Рассматриваемая функция имеет один аргумент, которым должна быть переменная типа char (*переменная_char*). При вызове функции get в эту переменную считывается следующий символ входного потока.

Синтаксис

входной_поток.get(*переменная_char*);

Пример

```
char next_symbol;
cin.get(next_symbol);
```

Если вы хотите воспользоваться функцией get для ввода данных из файла, замените объект cin именем файлового потока. Так, если входной файловый поток в программе называется in_stream, оператор

```
in_stream.get(next_symbol);
```

считает из файла один символ и поместит его в переменную next_symbol типа char.

Прежде чем воспользоваться функцией get для ввода данных из файла, программа должна соединить с ним поток посредством вызова функции open.

Функция-член putback (факультативный материал)

Иногда программе нужно знать, какой следующий символ находится во входном потоке. Причем после его чтения может оказаться, что обрабатывать его нет необходимости, и вам необходимо будет вернуть его обратно во входной поток. Например, если программа должна прочитать данные до первого пробела или символа перевода строки, не включая этот символ, ей придется прочитать его, чтобы узнать, что пора остановиться — и после этого его уже не будет во входном потоке.

Но какой-нибудь другой части программы может потребоваться прочитать и обработать этот символ. Существует множество способов решения данной проблемы — простейший из них заключается в использовании функции putback, которая является членом любого входного потока. Она принимает один аргумент типа char и помещает его значение во входной поток. Этим аргументом может быть любое выражение, возвращающее значение типа char.

Так, следующий код считывает символы из файла, соединенного со входным потоком fin, и записывает их в файл, соединенный с выходным потоком fout, он читает символы до первого пробела, не включая сам пробел:

```
fin.get(next);
while (next != ' ')
{
    fout.put(next);
    fin.get(next);
}
fin.putback(next);
```

После выполнения этого кода прочитанный пробел остается во входном потоке fin, поскольку программа, прочитав, возвращает его обратно.

Обратите внимание, что функция putback помещает символ во входной поток, а функция put — в выходной.

Символ, помещаемый функцией putback во входной поток, не обязательно должен быть последним считанным программой символом. Собственно говоря, это может быть произвольный символ. Если входной поток соединен с файлом, и вы помещаете в него некоторый символ с помощью функции putback, данный символ может быть прочитан программой из потока, но при этом он не будет помещен во входной файл, хотя программе «кажется», что он там находится.

Функция-член put

Каждый выходной поток имеет функцию-член put. У нее один аргумент, которым должно быть выражение типа char. При вызове данной функции значение ее аргумента (*выражение_char*) выводится в выходной поток.

Синтаксис

выходной_поток.put(*выражение_char*);

Пример

```
cout.put(next_symbol);
cout.put('a');
```

Если вы хотите воспользоваться функцией put для вывода данных в файл, замените объект cout именем файлового потока. Так, если выходной файловый поток в программе называется out_stream, оператор

```
out_stream.put('Z');
```

выведет символ Z в файл, соединенный с этим потоком.

Прежде чем воспользоваться функцией put для вывода данных в файл, программа должна соединить с ним поток путем вызова функции open.

Пример: проверка входных данных

Если пользователь вводит неверные данные, вся работа программы теряет смысл. Поэтому нужно вовремя проверять, что ввел пользователь, и давать ему возможность повторить ввод при обнаружении ошибки. В листинге 5.6 приведена функция get_int, предлагающая пользователю подтвердить, что введенные им данные правильны, и позволяющая повторить ввод, если это не так. Программа, в состав которой входит данная функция, предназначена для демонстрации, но сама функция (с небольшими изменениями) может использоваться практически в любой программе, принимающей ввод с клавиатуры.

Обратите внимание на вызов функции new_line, которая считывает все символы до конца текущей строки, но ничего с ними не делает. Она просто позволяет программе проигнорировать оставшуюся часть строки. Если пользователь введет No, программа считает первую букву (в данном случае N), затем вызовет функцию new_line, и та удалит из входного потока оставшуюся часть строки. А когда пользователь введет в следующей строке 75, программа сразу считает значение 75, а не букву o, оставшуюся от No. Если бы программа не содержала вызова new_line, следующим прочитанным ею значением в примере было бы не число 75, а буква o.

Кроме того, внимательно посмотрите на логическое выражение, которым завершается цикл do...while в функции get_int. В случае ввода неправильных данных после их прочтения пользователь наберет слово No (или no), после чего будет выполнена еще одна итерация цикла. Однако вместо того чтобы проверять, является ли первая буква введенного пользователем слова буквой N, программа выполняет проверку на ее не равенство букве Y (и на не равенство той же букве в нижнем регистре, то есть y). Если пользователь не сделает ошибки при вводе данных и затем наберет либо Yes либо No в любом регистре, то будет неважно, с каким из двух значений сравнивать введенное им слово. Но пользователь может ведь ввести и что-нибудь другое, так что надежнее всего выполнить проверку на равенство введенной строки значению Yes, поскольку в этом случае он действительно подтверждает правильность введенных перед этим данных. Если же данные неправильны, а он вместо No случайно ввел, например Bo (поскольку буквы B и N расположены на клавиатуре рядом), программа увидит, что первая буква введенного слова не равна 'Y', воспримет его ответ как No и повторит выполнение тела цикла, предоставив пользователю возможность еще раз ввести данные.

Ну а что произойдет, если пользователь допустит опечатку при вводе слова Yes и введет что-то, начинающееся не с буквы Y, а с другой буквы? Да ничего страшного. Тело цикла будет выполнено еще раз, пользователь повторно введет данные и подтвердит их правильность — на этот раз, скорее всего, уже без ошибок. Выполнение пользователем лишней работы все же лучше, чем обработка программой неверных данных, так что эта ошибка не будет иметь серьезных последствий.

Листинг 5.6. Проверка ввода

```
// Программа, демонстрирующая возможности функций new_line и get_input.
#include <iostream>
using namespace std;
void new_line();
```

```
// Удаляет остаток строки из входного потока,
// включая и символ '\n'.
// Эта версия функции принимает ввод только с клавиатуры.

void get_int(int& number);
// Постусловие: переменной number присвоено
// одобренное пользователем значение.

int main()
{
    int n;

    get_int(n);
    cout << "Final value read in = " << n << endl
         << "End of demonstration.\n";
    return 0;
}

// Используем библиотеку классов iostream.
void new_line()
{
    char symbol;
    do
    {
        cin.get(symbol);
    } while (symbol != '\n');
}

// Используем библиотеку классов iostream.
void get_int(int& number)
{
    char ans;
    do
    {
        cout << "Enter input number: ";
        cin >> number;
        cout << "You entered " << number
             << "Is that correct? (yes/no): ";
        cin >> ans;
        new_line();
    } while ((ans != 'Y') && (ans != 'y'));
}
```

Пример диалога

```
Enter input number: 57
You entered 57 Is that correct? (yes/no): No
Enter input number: 75
You entered 75 Is that correct? (yes/no): yes
Final value read in = 75
End of demonstration.
```

Ловушка: лишний символ '\n' во входном потоке

При использовании функции-члена get следует иметь в виду, что входные данные могут содержать любые символы, включая пробелы и символы перевода строки.

Между тем программисты нередко забывают удалить из входного потока символ '\n', которым завершается каждая строка. А если этот символ не прочитан (и таким образом не удален из потока), при следующей попытке прочитать «настоящий» символ с помощью функции get программа вместо него получит символ '\n'. Для очистки входного потока от оставшихся символов перевода строки можно воспользоваться функцией new_line, приведенной в листинге 5.6. Рассмотрим конкретный пример.

Вполне допустимо смешивать различные способы ввода данных из потока cin. Скажем, следующий код:

```
cout << "Enter a number:\n";
int number;
cin >> number;
cout << "Now enter a letter:\n";
char symbol;
cin.get(symbol);
```

является вполне корректным.

Однако, смещение:

```
Enter a number:
21
Now enter a letter:
A
```

может вызвать определенные проблемы

Значением переменой number в результате, как и ожидалось, будет 21. Но если вы полагаете, что значением переменной symbol станет A, то ошибаетесь. Эта переменная получит значение \n. После чтения числа 21 следующим символом во входном потоке будет '\n' — его и считает функция cin.get(symbol). Вспомните, что функция get не пропускает символы перевода строки и пробелы. (Собственно говоря, в зависимости от того, какой код следует в программе далее, пользователю может вообще не представиться возможности ввести букву A. После того, как в переменную symbol будет помещен символ '\n', программа перейдет к следующему оператору, и если он выводит результаты на экран, они появятся там раньше, чем пользователь сможет ввести букву A.)

Ниже приводятся еще два варианта кода, который считывает в переменную number число 21, а в переменную symbol — символ 'A'. Вот первый из них:

```
cout << "Enter a number:\n";
int number;
cin >> number;
cout << "Now enter a letter:\n";
char symbol;
cin >> symbol;
```

В качестве альтернативы можно воспользоваться функцией new_line, определенной в программе листинга 5.6:

```
cout << "Enter a number:\n";
int number;
cin >> number;
new_line();
```

```
cout << "Now enter a letter:\n";
char symbol;
cin.get(symbol);
```

Как видно из второго варианта кода, в программе можно смешивать два способа ввода данных из потока cin, но делать это следует очень внимательно.

Упражнения для самопроверки

22. Предположим, что c — переменная типа char. Какая разница между следующими двумя операторами:

    ```
    cin >> c;
    ```

    и

    ```
    cin.get(c);
    ```

23. Предположим, что c — переменная типа char. Какая разница между следующими двумя операторами:

    ```
    cout << c;
    ```

    и

    ```
    cout.put(c);
    ```

24. (Вопрос для тех, кто прочитал дополнительный раздел о функции putback.) Функция-член putback помещает заданный символ во входной поток. Должен ли передаваемый ей символ быть последним прочитанным из данного потока символом? Например, если программа считывает из входного потока символ 'a', может ли функция putback поместить во входной поток символ 'b'?

25. Рассмотрим следующий код (и предположим, что он выполняется в составе программы, не содержащей ошибок):

    ```
    char c1, c2, c3, c4;
    cout << "Enter a line of input:\n";
    cin.get(c1);
    cin.get(c2);
    cin.get(c3);
    cin.get(c4);
    cout << c1 << c2 << c3 << c4 << "END OF OUTPUT";
    ```

    Если пользователь ввел такие данные:

    ```
    Enter a line of input:
    a b c d e f g
    ```

    какова будет следующая выведенная этим кодом строка?

26. Рассмотрим следующий код (и предположим, что он выполняется в составе программы, не содержащей ошибок):

    ```
    char next;
    int count = 0;
    cout << "Enter a line of input:\n";
    cin.get(next);
    while (next != '\n')
    {
        if ((count%2) == 0)   // true, если значение переменной count четное.
    ```

```
        cout << next;
    count++;
    cin.get(next);
}
```

Если пользователь ввел указанные данные:

```
Enter a line of input:
abcdef gh
```

какова будет следующая выведенная этим кодом строка?

27. Предположим, что выполняется программа, описанная в упражнении 26. Если пользователь ввел такие данные:

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
```

какова будет следующая строка, выведенная этой программой?

28. Рассмотрим следующий код (и предположим, что он выполняется в составе программы, не содержащей ошибок):

```
char next;
int count = 0;
cout << "Enter a line of input:\n";
cin >> next;
while (next != '\n')
{
    if ((count%2) == 0)   // Истинно, если значение переменной count четное.
        cout << next;
    count++;
    cin >> next;
}
```

Если введены данные

```
Enter a line of input
0 1 2 3 4 5 6 7 8 9 10 11
```

какова будет следующая строка, выведенная этим кодом?

Функция-член eof

Каждый файловый входной поток имеет функцию-член `eof`, с помощью которой можно определить, что все содержимое файла уже прочитано и больше вводить нечего.

Название функции `eof` представляет собой сокращение от англ. end of file – конец файла. У этой функции нет аргументов, и если входной поток называется, скажем, `fin`, вызов функции `eof` записывается так:

```
fin.eof()
```

Это логическое выражение, которое может использоваться для управления циклом `while`, `do...while` или оператором `if...else`. Если выражение истинно, это означает, что программа достигла конца файла; если же в файле остались непрочитанные данные, выражение ложно.

Поскольку, как правило, в программе выполняется проверка на не конец файла, вызов функции eof обычно предваряется оператором отрицания (!). Рассмотрим в качестве примера следующий оператор:

```
if (!fin.eof())
    cout << "Not done yet.";
else
    cout << "End of the file.";
```

Логическое выражение после ключевого слова if означает «не конец файла, соединенного с потоком fin». Поэтому, если программа еще не достигла конца файла, соединенного с потоком fin, приведенный оператор выведет на экран следующее:

```
Not done yet.
```

Если же программа достигла конца файла, этот оператор выведет:

```
End of the file.
```

Теперь предположим, что входной поток in_stream подключен к файлу посредством вызова функции open. Тогда содержимое файла может быть выведено на экран с помощью следующего цикла while:

```
in_stream.get(next);
while (! in_stream.eof())
{
    cout << next;     // Если хотите, можете заменить этот
                      // оператор функцией cout.put(next).
    in_stream.get(next);
}
```

Этот цикл while по очереди считывает в переменную next типа char символы из файла и выводит их на экран. Когда программа достигает конца файла, значение выражения in_stream.eof() становится равным true. Поэтому выражение

```
(! in_stream.eof())
```

принимает значение false и цикл завершается.

Обратите внимание, что выражение in_stream.eof() не равно true до тех пор, пока программа не попытается прочитать символ, находящийся после конца файла. Предположим, что файл содержит следующий текст (символ c в нем последний — за ним не следует символ перевода строки):

```
ab
c
```

На самом деле это следующие четыре символа:

```
ab<символ перевода строки '\n'>c
```

Приведенный выше цикл прочитает и выведет на экран символ 'a', затем — символ 'b' , далее прочитает и выведет на экран символ перевода строки '\n', а потом — символ 'c'. К этому моменту цикл прочитает все имеющиеся в файле символы, но выражение in_stream.eof() по-прежнему будет возвращать false до тех пор, пока программа не попытается прочитать еще один символ. Вот почему в нашем цикле функция in_stream.get(next) стоит в конце. После вывода последнего

имеющегося в файле символа программа еще раз выполнит этот оператор, и будет осуществлен переход к началу цикла и проверка его условия. Только тогда условие окажется ложным и выполнение цикла завершится.

В конце каждого файла имеется специальный маркер конца файла. Функция-член eof возвращает true только после прочтения программой этого маркера. Поэтому приведенный выше цикл while может прочитать еще один символ после того, как будет прочитан последний символ файла. Однако маркер конца файла — это не обычный символ, и с ним нельзя обращаться, как с обычным символом (например, если вывести его на экран, результат будет непредсказуемым). Система помещает этот маркер в конец каждого файла автоматически.

Функция eof используется в примере программы, который приведен в следующем разделе.

Теперь вам известны два способа определения конца файла. Один из них заключается в применении функции eof, а второй описан в разделе «Совет программисту: проверка на конец файла». Обычно можно прибегать к любому из этих методов, но большинство программистов пользуются ими в разных ситуациях. Если у вас нет личных предпочтений, придерживайтесь следующего правила: функцией eof пользуйтесь в тех случаях, когда входные данные интерпретируются как текст и вводятся с помощью функции-члена get, а второй метод применяйте при обработке числовых данных.

Упражнения для самопроверки

29. Предположим, ins — входной файловый поток, соединенный с файлом посредством функции-члена open. Допустим, программа только что прочитала последний символ данного файла. Какое значение — true или false — вернет вызов ins.eof() в этой точке выполнения программы?
30. Напишите определение функции типа void с именем text_to_screen и одним формальным параметром file_stream типа ifstream. Предусловие и постусловие этой функции таковы:

```
// Предусловие: поток file_stream соединен с файлом
// посредством вызова функции-члена open.
// Постусловие: содержимое файла, соединенного с потоком file_stream,
// в точности отображено на экране.
// (Функция не закрывает файл.)
```

Пример: редактирование текстового файла

Описанная здесь программа (листинг 5.7) представляет собой очень простой пример автоматизированного редактирования текстового файла. Такая программа могла бы использоваться компанией-производителем программного обеспечения для обновления рекламной литературы. Предположим, что эта компания выпускает компиляторы для языка программирования C и недавно анонсировала новую линию компиляторов языка C++. Используя нашу программу, компания на основе рекламных материалов компилятора C может сгенерировать точно такие же рекламные материалы для C++. Программа считывает входные данные из файла, который

включает описание достоинств компилятора C, и записывает в другой файл точно такой же текст с похвалами по адресу компилятора C++. Исходный файл называется cad.dat, а результирующий – cplusad.dat. Все символы исходного файла копируются без изменений за исключением заглавной буквы C, вместо которой программа записывает в результирующий файл строку "C++". Она предполагает, что буква C в исходном файле всегда является именем языка программирования C.

Обратите внимание, что при чтении символов из исходного файла в программе сохраняются разрывы строк. Символ перевода строки '\n' обрабатывается так же, как любой другой символ, – он считывается из исходного файла с помощью функции-члена get и записывается в результирующий файл посредством оператора вывода <<. Чтение именно с помощью функции get, а не оператора ввода >> выполняется потому, что если мы воспользуемся вторым, результирующий файл будет содержать сплошной массив символов без пробелов и разрывов строк, поскольку этот оператор пропускает символы пробела и перевода строки.

Обратите также внимание на применение функции-члена eof, с помощью которой определяется конец исходного файла для остановки выполнения цикла.

Листинг 5.7. Редактирование файла

```
// Программа создания файла cplusad.dat, аналогичного
// файлу cad.dat за исключением вхождений символа 'C',
// заменяемого строкой 'C++'.
// Предполагается, что буква C, набранная в  верхнем регистре,
// в файле cad.dat встречается только в качестве имени
// языка программирования C.

#include <fstream>
#include <iostream>
#include <cstdlib>
using namespace std;

void add_plus_plus(ifstream& in_stream, ofstream& out_stream);
// Предусловие: поток in_stream соединен с файлом cad.dat
// с помощью вызова функции open.
// Поток out_stream подключен к файлу cplusad.dat с помощью вызова функции open.
// Постусловие: содержимое файла, соединенного с потоком in_stream,
// скопировано в файл, соединенный с потоком out_stream, и при этом
// каждое вхождение 'C' заменено "C++".
// (Функция не закрывает файлы.)

int main()
{
    ifstream fin;
    ofstream fout;

    cout << "Begin editing files.\n";

    fin.open("cad.dat");
    if (fin.fail())
    {
        cout << "Input file opening failed.\n";
        exit(1);
    }
```

продолжение

Листинг 5.7 (*продолжение*)

```
    fout.open("cplusad.dat");
    if (fout.fail())
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }

add_plus_plus(fin, fout);


    fin.close();
    fout.close();

    cout << "End of editing files.\n";
    return 0;
}

void add_plus_plus(ifstream& in_stream, ofstream& out_stream)
{
    char next;

    in_stream.get(next);
    while (! in_stream.eof())
    {
        if (next == 'C')
            out_stream << "C++";
        else
            out_stream << next;

        in_stream.get(next);
    }
}
```

cad.dat
(Не изменяется программой)

```
C is one of the world's most modern
programming languages. There is no
language as versatile as C, and C
is fun to use.
```

cplusad.dat
(После выполнения программы)

```
C++ is one of the world's most modern
programming languages. There is no
language as versatile as C++, and C++
is fun to use.
```

Данные, выводимые на экран

```
Begin editing files.
End of editing files.
```

Предопределенные символьные функции

При обработке текста часто требуется преобразовать прописные буквы в строчные и наоборот. Для такого преобразования используется стандартная функция `toupper`. Например, `toupper('a')` возвращает `'A'`. Если аргумент функции `toupper` не является символом, набранном в нижнем регистре, она просто возвращает аргумент без изменений, так что `toupper('A')` возвращает `'A'`. Еще одна стандартная функция, `tolower`, выполняет обратное преобразование – буквы верхнего регистра в нижний.

Объявления функций toupper и tolower находятся в заголовочном файле cctype, поэтому любая программа, в которой они используются, должна содержать следующую директиву #include:

```
#include <cctype>
```

В таблице 5.9 приведено описание наиболее популярных функций из библиотеки cctype.

Таблица 5.2. Некоторые стандартные символьные функции, объявление которых содержится в заголовочном файле cctype

Функция	Описание	Пример
toupper(выражение_char)	Возвращает символ, являющийся значением аргумента выражение_char, переведенный в верхний регистр	char c = toupper('a'); cout << c; Возвращает: A
tolower(выражение_char)	Возвращает символ, являющийся значением аргумента выражение_char, переведенный в нижний регистр	char c = tolower('A'); cout << c; Возвращает: a

isupper(выражение_char)	Возвращает true, если значение аргумента выражение_char является буквой верхнего регистра; в противном случае возвращает false	if (isupper(c))
		  cout << c << " is uppercase.";
		else
		  cout << c << " is not uppercase.";

islower(выражение_char)	Возвращает true, если значение аргумента выражение_char является буквой нижнего регистра; в противном случае возвращает false	char c = 'a'; if (islower(c)) cout << c << " is lowercase."; Возвращает: a is lowercase.

isalpha(выражение_char)	Возвращает true, если значение аргумента выражение_char является символом алфавита; в противном случае возвращает false	char c = '$';
		if (isalpha(c))
		  cout << c << " is a letter.";
		else
		  cout << c << " is not a letter.";
		Возвращает: $ is not a letter.

isdigit(выражение_char)	Возвращает true, если значение аргумента выражение_char является цифрой от 0 до 9; в противном случае возвращает false	if (isdigit('3'))
		  cout << "It's a digit.";
		else
		  cout << "It's not a digit.";
		Возвращает: It's a digit.

продолжение

Таблица 5.2 (*продолжение*)

Функция	Описание	Пример

`isspace(выражение_char)`	Возвращает `true`, если значение аргумента `выражение_char` является символом пропуска, таким как пробел или символ перевода строки; в противном случае возвращает `false`	`if (isspace(' '))`
		`  cout << "It's a space symbol.";`
		`else`
		`  cout << "It's not a space symbol.";`
		Возвращает: `It's not a space symbol.`

Функция `isspace` возвращает `true`, если ее аргумент является символом пропуска (то есть символом, представляемым на экране пустым пространством): пробелом, символом табуляции или перевода строки. Если аргументом функции `isspace` не является символ пропуска, она возвращает `false`. Таким образом, функция `isspace(' ')` возвращает `true`, а `isspace('a')` возвращает `false`. Приведенный ниже код считывает последовательность символов, завершаемую точкой, и выводит ее на экран, заменив все символы пропуска символом `'-'` :

```
char next;
do
{
    cin.get(next);
    if (isspace(next))   // Истинно, если next содержит символ пропуска.
        cout << '-';
    else
        cout << next;
} while (next != '.');
```

Например, если ввести

```
Ahh   do be do.
```

он выведет на экран следующее:

```
Ahh---do-be-do.
```

Ловушка: функции toupper и tolower возвращают значения типа int

C++ может интерпретировать символы как целые числа типа `int`. Каждому символу в этом языке назначено число, и при записи символа в переменную типа `char` в память компьютера, выделенную для этой переменой, на самом деле записывается соответствующее число. Со значением типа `char` тоже можно работать как с числом, например присвоить его переменной типа `int`. Допустимо и обратное действие — запись значения переменной типа `int` в переменную типа `char` (но при условии, что число не слишком велико). То есть, тип данных `char` может использоваться для работы как с символами, так и с небольшими целыми числами.

В большинстве случаев переменные типа `char` можно считать символами и не беспокоиться об этой их особенности. Но при работе с функциями из библиотеки `cctype` эта особенность может быть важна. В частности, функции `toupper` и `tolower`

на самом деле возвращают значения типа int, а не char; иными словами, они возвращают число, соответствующее символу, полученному в результате преобразования, а не сам этот символ. Поэтому следующий оператор выведет не символ A, а соответствующее ей число:

```
cout << toupper('a');
```

Чтобы компьютер интерпретировал значение, возвращенное функцией toupper или tolower, как символ, а не как число, нужно указать, что вам требуется значение типа char. Для этого можно поместить возвращенное функцией значение в переменную типа char. Следующий код выведет на экран символ A:

```
char c = toupper('a');   // Помещает символ A в переменную c.
cout << c;
```

Можно также выполнить явное приведение значения, возвращенного функцией toupper или tolower, к типу данных char:

```
cout << static_cast<char>(toupper('a'));   // Выводит символ A.
```

Упражнения для самопроверки

31. Рассмотрим следующий код (и предположим, что он выполняется в составе программы, не содержащей ошибок):

```
cout << "Enter a line of input:\n";
char next;
do
{
    cin.get(next);
    cout << next;
} while ((! isdigit(next)) && (next != '\n'));
cout << "<END OF INPUT";
```

Если ввод данных пользователем начинается таким образом:

```
Enter a line of input:
I'll see you at 10:30 AM.
```

какой будет следующая выведенная этим кодом строка?

32. Напишите код C++, считывающий строку текста и выводящий ее на экран, предварительно удалив из строки все буквы, набранные в верхнем регистре.

5.4. Наследование

Самая мощная возможность C++ — *наследование классов*. Когда мы говорим, что некоторый класс *наследует* другой класс или является *производным* от другого класса, то имеется в виду, что он создан на основе этого класса путем добавления элементов. Например, класс файловых входных потоков наследует класс всех входных потоков, и в него добавлены функции-члены open и close. Поток cin принадлежит классу всех входных потоков, но не принадлежит классу файловых входных потоков, поскольку не имеет функций-членов open и close. Если объявить в программе поток типа ifstream, это будет файловый входной поток, поскольку он будет содержать функции-члены open и close.

В данном разделе понятие наследования рассматривается применительно к потокам. Подробно о наследования классов мы расскажем далее в этой книге, а пока изучите лишь базовый материал, необходимый для простейших действий с производными классами.

Наследование для потоковых классов

Для того чтобы освоиться с терминологией наследования, нужно вспомнить, что объект — это переменная, содержащая функции-члены, а класс — тип данных, переменные которого являются объектами. Потоки (такие, как cin, cout, входные файловые потоки и выходные файловые потоки) — это объекты, а потоковые типы (например, ifstream и ofstream) — это классы. Учитывая вышесказанное, рассмотрим несколько примеров потоков и потоковых классов.

И предопределенный поток cin, и объявленный в программе файловый входной поток являются входными потоками, поэтому между ними много общего. Так, с любым входным потоком можно использовать оператор ввода >>. С другой стороны, файловый входной поток можно соединить с файлом посредством функции-члена open, а у потока cin такой функции нет. Таким образом, эти два потока схожи, но все же относятся к разным типам — входной файловый поток имеет тип ifstream, а поток cin, как будет рассказано далее, принадлежит к типу istream (без буквы f). Классы ifstream и istream — разные, но тесно связанные между собой типы, а именно: класс ifstream является *производным* от класса istream. Сейчас мы подробно объясним, что это означает.

Рассмотрим функцию, считывающую два целых числа из входного потока source_file и выводящую на экран их сумму:

```
void two_sum(ifstream& source_file)
{
    int n1, n2;
    source_file >> n1 >> n2;
    cout << n1 << " + " << n2 << " = " << (n1 + n2) << endl;
}
```

Допустим, что программа содержит это определение функции и следующее объявление потока:

```
ifstream fin;
```

Если поток fin соединен с файлом посредством функции open, можно воспользоваться функцией two_sum для чтения из этого файла двух чисел и вывода их суммы. Вызов функции two_sum будет таким:

```
two_sum(fin);
```

Теперь предположим, что далее в той же программе нужно прочитать два числа, введенных с клавиатуры, и вывести на экран их сумму. Поскольку все входные потоки сходны, можно подумать, что для выполнения этой задачи следует снова вызвать функцию two_sum, передав ей поток cin:

```
two_sum(cin);   // НЕ СРАБОТАЕТ
```

Как показывает комментарий, при компиляции программы приведенная строка вызовет сообщение об ошибке. Объект cin не относится к типу ifstream, как того требует функция two_sum, он имеет тип istream. И если вы хотите передать этот поток функции two_sum, ее параметр должен быть объявлен как istream, а не как ifstream. Следующая версия функции two_sum примет cin в качестве аргумента:

```
void better_two_sum(istream& source_file)
{
    int n1, n2;
    source_file >> n1 >> n2;
    cout << n1 << " + " << n2 << " = " << (n1 + n2) << endl;
}
```

За исключением типа параметра функции better_two_sum и two_sum одинаковы. Поскольку объект cin соответствует типу параметра функции better_two_sum, он может быть ей передан:

```
better_two_sum(cin);
```

Ну а теперь хорошая новость: функция better_two_sum может использоваться с любым входным потоком, а не только с cin.

Следующий ее вызов:

```
better_two_sum(fin);
```

считается допустимым.

Такая запись, видимо, покажется вам необычной, ведь объект fin относится к типу ifstream, а аргумент функции better_two_sum должен иметь тип istream, то есть складывается впечатление, что объект fin относится к обоим названным типам. Это действительно так, поскольку данные типы связаны определенным отношением: класс ifstream является производным от класса istream.

Когда мы говорим, что класс A является производным от класса B, это означает, что класс A содержит все элементы класса B плюс некоторые дополнительные элементы. Так, с потоками обоих типов можно использовать оператор вывода >>, однако у потока ifstream имеются и дополнительные элементы, поэтому с ним можно производить больше операций, чем с потоком istream. Например, одним из дополнительных элементов потока ifstream является функция open. Поток cin относится к типу istream и не относится к типу ifstream, поэтому для него нельзя вызвать функцию open.

Любой поток типа ifstream является также и потоком типа istream, так что его можно передавать функции, формальный параметр которой имеет тип istream. Так что если функция должна работать с входным потоком, объявление ее параметра как istream делает ее более универсальной. Такой функции можно передавать как поток cin, так и входной поток, соединенный с файлом.

Идея наследования на первый взгляд может показаться странной, но на самом деле в ней нет ничего необычного. Чтобы ее понять, рассмотрим пример из повседневной жизни. Представьте себе класс автомобилей с откидным верхом, производный от класса автомобилей. Каждый автомобиль с откидным верхом является автомобилем, но имеет особенности, отсутствующие у автомобилей других типов: его можно превратить в открытую машину (можно сказать, что у него имеется дополнительная функция open). Точно так же класс ifstream входных файловых

потоков является производным от класса istream, состоящего из всех входных потоков. Каждый входной файловый поток является входным потоком, но снабжен дополнительными свойствами, которые отсутствуют у других входных потоков (например, у потока cin).

Для производных классов часто используется метафора «наследование» или «семейные отношения». Если класс B является производным от класса A, он называется *дочерним* по отношению к классу A, а тот в свою очередь именуется *родительским* по отношению к классу B. Производный класс наследует функции-члены родительского класса. Скажем, каждый автомобиль с откидным верхом наследует от класса всех автомобилей четыре колеса, а каждый входной файловый поток наследует от класса входных потоков оператор >>. Вот поэтому технологию создания и использования производных классов называют наследованием.

Если вы пока не освоились с понятием производных классов, не беспокойтесь, все станет ясно при выполнении практической работы. В следующей врезке еще раз кратко повторяются сведения, необходимые для использования производных классов, рассматриваемых в этом разделе.

Как сделать потоковые параметры универсальными

Если вы хотите определить функцию, принимающую в качестве аргумента входной поток, и в одних случаях этим аргументом должен быть cin, а в других — файловый поток, то формальный параметр функции должен иметь тип istream. Однако входной файловый поток, даже используемый в качестве аргумента типа istream, должен быть объявлен с типом ifstream.

Аналогично если вы хотите определить функцию, принимающую в качестве аргумента выходной поток, и в одних случаях этим аргументом должен быть cout, а в других — файловый поток, формальный параметр этой функции должен иметь тип ostream. Однако файловый выходной поток, даже используемый в качестве аргумента типа ostream, должен быть объявлен с типом ofstream.

Кроме того, имейте в виду, что потоковый параметр типа istream или ostream в функции нельзя ни открывать, ни закрывать. Поэтому соответствующий объект должен быть открыт до вызова функции и закрыт после него.

До сих пор мы говорили о двух классах входных потоков: istream и производном от него ifstream. Таким же отношением связаны два класса выходных потоков. Все выходные потоки представлены классом ostream, и к нему относится объект cout. Файловые выходные потоки представлены классом ofstream, производным от класса ostream. Следующая функция выводит слово Hello в выходной поток, переданный ей в качестве аргумента:

```
void say_hello(ostream& any_out_stream)
{
    any_out_stream << "Hello";
}
```

Первый из следующих вызовов выводит слово Hello на экран, а второй записывает его в файл afile.dat:

```
ofstream fout;
fout.open("afile.dat");
```

```
say_hello(cout);
say_hello(fout);
```

Обратите внимание, что выходной файловый поток имеет тип ofstream и в то же время принадлежит к типу ostream.

Пример: еще одна функция new_line

В качестве еще одного примера того, как сделать потоковую функцию более универсальной, рассмотрим функцию new_line, определенную в программе листинга 5.6. Эта функция обрабатывает только ввод данных с клавиатуры, то есть из предопределенного потока cin, и у нее отсутствуют аргументы. Ниже приведена еще одна ее версия с формальным параметром типа istream, в котором задается входной поток:

```
// Используем библиотеку классов iostream:
void new_line(&istream in_stream)
{
    char symbol;
    do
    {
        in_stream.get(symbol);
    } while (symbol != '\n');
}
```

Теперь предположим, что программа содержит эту новую версию функции. Когда она принимает данные из файлового входного потока fin, вызов

```
new_line(fin);
```

удаляет из него остаток текущей строки.

Если в другом месте программа принимает данные, которые вводятся с клавиатуры, вызов

```
new_line(cin);
```

удаляет из потока cin остаток текущей строки.

В случае использования в программе только переработанной версии функции new_line, принимающей потоковый аргумент, ей всегда нужно будет передавать имя потока, даже если в этот момент программа принимает данные, вводимые с клавиатуры. Но благодаря поддержке языком C++ технологии перегрузки в одной программе может быть две версии функции new_line: одна без аргументов, приведенная в листинге 5.6, а вторая с аргументами — та, которую мы только что определили. В программе, содержащей оба определения функции new_line, следующие два вызова эквивалентны:

```
new_line(cin);
```

и

```
new_line();
```

Наличие двух функций new_line в программе не обязательно, поскольку версия с одним аргументом полностью удовлетворяет ее нуждам. Но многие программисты предпочитают иметь и версию без аргументов для ввода данных с клавиатуры, поскольку такой вид ввода используется очень часто.

Аргументы функции, используемые по умолчанию (факультативный материал)

Вместо того чтобы создавать две версии функции new_line, можно определить в ней *аргумент, используемый по умолчанию*. Ниже приведена третья версия этой функции, в которой задан такой аргумент.

```
// Используем библиотеку классов iostream:
void new_line(&istream in_stream = cin)
{
    char symbol;
    do
    {
        in_stream.get(symbol);
    } while (symbol != '\n');
}
```

Если вызвать данную функцию таким образом:

```
new_line();
```

формальному параметру по умолчанию будет присвоен аргумент cin. Если же вызвать ее так:

```
new_line(fin);
```

формальному параметру будет присвоен аргумент fin. Этот способ определения аргументов может использоваться с любыми типами и произвольным количеством параметров.

Когда значения по умолчанию определены только для части параметров, параметры, для которых они заданы, должны располагаться подряд в конце списка параметров. В вызове этой функции должно быть задано как минимум количество аргументов, соответствующее числу формальных параметров, не имеющих значений по умолчанию. Кроме того, можно задать аргументы и для остальных параметров, но разумеется, их количество не должно превышать общее число параметров функции. Например:

```
// Тестирование поведения аргументов,
// применяемых по умолчанию.
// Используем библиотеку классов iostream.
void default_args(int arg1, int arg2, int arg3 = -3,
                  int arg4 = -4)
{
    cout << arg1 << ' ' << arg2 << ' ' << arg3 << ' ' << arg4
         << endl;
}
```

Эту функцию можно вызывать с двумя, тремя или четырьмя аргументами. Так, в вызове функции

```
default_args(5, 6);
```

заданы только те аргументы, для которых не определены значения по умолчанию. Она выводит следующее данные:

```
5 6 -3 -4
```

Теперь рассмотрим такой вызов функции:

```
default_args(6, 7, 8);
```

В нем заданы два аргумента, для которых не определены значения по умолчанию, и еще один аргумент, для которого значение по умолчанию определено. Эта функция возвращает следующие числа:

```
6 7 8 -4
```

А в вызове функции

```
default_args(5, 6, 7, 8);
```

заданы все четыре аргумента, и она выводит следующее:

```
5 6 7 8
```

Упражнения для самопроверки

33. К какому типу относится поток cin? К какому типу относится поток cout?
34. Определите функцию с именем copy_char, принимающую единственный аргумент – входной поток. Эта функция считывает из входного потока один символ и выводит его на экран. В качестве аргумента ей может передаваться как cin, так и входной файловый поток (во втором случае поток должен быть соединен с файлом до вызова функции, чтобы последней не пришлось открывать или закрывать файл). Например, первый из двух следующих вызовов функции copy_char выводит на экран символ из файла stuff.dat, а второй – символ, введенный с клавиатуры:

```
ifstream fin;
fin.open("stuff.dat");
copy_char(fin);
copy_char(cin);
```

35. Определите функцию с именем copy_line, принимающую единственный аргумент – входной поток. Эта функция считывает из входного потока одну строку и выводит ее на экран. В качестве аргумента ей может передаваться как cin, так и входной файловый поток (во втором случае поток должен быть соединен с файлом до вызова функции, чтобы последней не пришлось открывать или закрывать файл). Например, первый из двух следующих вызовов copy_line выводит на экран строку из файла stuff.dat, а второй – строку, введенную с клавиатуры:

```
ifstream fin;
fin.open("stuff.dat");
copy_line(fin);
copy_line(cin);
```

36. Определите функцию с именем send_line, принимающую единственный аргумент – входной поток. Эта функция считывает с клавиатуры одну строку и выводит ее в выходной поток, заданный в качестве аргумента. Ей может передаваться как cout, так и выходной файловый поток (во втором случае поток должен быть соединен с файлом до вызова функции, чтобы последней не

пришлось открывать или закрывать файл). Например, в первом из двух следующих вызовов функция send_line записывает одну введенную с клавиатуры строку в файл morestuff.dat, а вторым выводит другую введенную с клавиатуры строку на экран:

```
ofstream fout;
fout.open("morestuff.dat");
cout << "Enter 2 lines of input:\n";
send_line(fout);
send_line(cout);
```

37. (Упражнение для тех, кто изучил факультативный раздел об аргументах, используемых по умолчанию.) Что выведет функция

```
void func(double x, double y = 1.1, double z = 2.3)
{
    cout << x << " " << y <<  " " << z << endl;
}
```

в ответ на вызовы

```
func(2.0);
func(2.0, 3.0);
func(2.0, 3.0, 4.0);
```

38. (Упражнение для тех, кто изучил факультативный раздел об аргументах, используемых по умолчанию.) Напишите несколько функций, перегружающих одно и то же имя функции для достижения того же эффекта, что в вызовах из предыдущего упражнения, без использования значений по умолчанию.

Резюме

- Поток типа ifstream можно соединить с файлом посредством вызова функции-члена open. После этого программа может получать данные из файла.
- Поток типа ofstream можно соединить с файлом, используя вызов функции-члена open. После этого программа может выводить данные в файл.
- Для проверки успешного выполнения вызова функции open можно применить функцию-член fail.
- Объект — это переменная, с которой связаны функции. Такие функции называются функциями-членами. Класс представляет собой тип, переменные которого являются объектами. Пример объекта — поток, примеры классов — типы ifstream и ofstream.
- Вызов функции-члена объекта имеет следующий синтаксис:

  вызывающий_объект.имя_функции_члена(список_аргументов);

  Вот пример вызова функции-члена precision для объекта-потока cout:

  ```
  cout.precision(2);
  ```
- Для форматирования вывода могут использоваться функции-члены потоков, такие как width, setf и precision. Они действуют одинаково как для предобъявленного потока cout, так и для объявленного в программе потока, соединенного с файлом.

- Каждый входной поток имеет функцию-член get, предназначенную для возврата одного введенного символа. Эта функция не пропускает ни пробелы, ни символы табуляции, ни символы перевода строки. А каждый выходной поток имеет функцию-член put, с помощью которой можно записать один символ в выходной поток.
- Используя функцию-член eof, можно узнать, когда программа достигла конца файла. Данная функция полезна для обработки текста, а при обработке числовых данных удобнее определять конец файла с помощью проверки на не конец файла, описанной в этой главе.
- Функция может иметь формальные параметры потокового типа, но их нужно передавать по ссылке, а не по значению. Для входного файлового потока должен использоваться параметр типа ifstream, а для выходного — параметр типа ofstream. (Другие возможности перечислены в следующем пункте.)
- Если потоковый параметр функции имеет тип istream, соответствующий аргумент может быть либо потоком cin, либо входным файловым потоком типа ifstream. Если потоковый параметр функции имеет тип ostream, соответствующий аргумент может быть либо потоком cout, либо выходным файловым потоком типа ofstream.
- Для параметра функции в ее определении может быть задано значение по умолчанию на тот случай, когда соответствующий аргумент не определен в ее вызове. В заголовке функции параметры, для которых заданы значения по умолчанию, должны следовать за параметрами, для которых такие значения не заданы. Аргументы, не имеющие значений по умолчанию, обязательно должны быть определены в вызове функции, а за ними могут быть заданы все или часть остальных аргументов.

Ответы к упражнениям для самопроверки

1. Потоки fin и fout объявляются таким образом:

```
ifstream fin;
ofstream fout;
```

   В начале программного кода располагается следующая директива #include:

```
#include <fstream>
```

   Кроме того, в программе необходима директива

```
using namespace std;
```

2.
```
fin.open("stuff1.dat");
if (fin.fail())
{
    cout << "Input file opening failed.\n";
    exit(1);
}
fout.open("stuff2.dat");
```

```
if (fout.fail())
{
    cout << "Output file opening failed.\n";
    exit(1);
}
```

3. `fin.close();`
   `fout.close();`

4. Для этого необходимо заменить поток `out_stream` потоком `cout`. Обратите внимание, что поток `cout` объявлять не нужно, как не требуется открывать его с помощью вызова функции `open` и закрывать посредством вызова функции `close`.

5. `#include <cstdlib>`

   Кроме того, в программе нужна следующая директива:

   ```
   using namespace std;
   ```

6. Функция `exit` возвращает свой аргумент операционной системе. По стандартному соглашению система интерпретирует значение `1` как указание на ошибку, а значение `0` как указание на успешное завершение программы.

7. `bla.dobedo (7);`

8. Как в файлах, так и в переменных программы могут храниться значения, причем и из тех, и из других эти значения могут считываться для использования в программе. Но переменные существуют только до тех пор, пока выполняется программа, а файлы остаются в неприкосновенности и после ее завершения либо могут уже существовать до ее запуска. Иными словами, файлы постоянны, а переменные — временны. Кроме того, в файлах могут храниться большие объемы данных, тогда как переменные не настолько вместительны.

9. До сих пор мы встречались с функциями-членами `open`, `close` и `fail`. Вот пример их использования:

```
int c;
ifstream in;
ofstream out;
in.open("in.dat");
if (in.fail())
{
   cout << "Input file opening failed.\n";
   exit(1);
}
in >> c;

out.open("out.dat");
if (out.fail())
{
    cout << "Output file opening failed.\n";
    exit(1);
}
out << c;

in.close();
out.close();
```

10. Об этом рассказывалось в начале главы. Файл должен быть закрыт и открыт еще раз. В результате текущая позиция чтения вновь окажется в начале файла и можно будет начать чтение сначала.

11. Это внешнее имя файла и имя потока. Внешнее имя файла используется операционной системой и является настоящим именем файла, но в программе применяется только один раз в вызове функции open, соединяющей файл с потоком. Имя потока — это имя потоковой переменной (обычно типа ifstream или ofstream). После вызова функции программа всегда использует в качестве имени файла имя потока.

12.
```
*  123*123*
*  123*123*
```

Каждый из пропусков содержит два пробела. Обратите внимание, что вызовы функции width и модификатора setw воздействуют только на один выводимый элемент.

13.
```
*  123*123  *  123*
```

Каждый из пропусков содержит два пробела.

14.
```
*  123*123*
* +123*+123*
*123  *123  *
```

Во второй строке между символами * и + имеется один пробел, остальные пропуски содержат по два пробела.

15. В файл stuff.dat будет выведено то же самое, что и на экран в упражнении 14.

16.
```
*12345*
```

Обратите внимание, что выведено все число, хотя оно содержит больше символов, чем указано в вызове setw.

17. а) ios::fixed. Установка флага вызывает вывод чисел с плавающей запятой в стандартном десятичном формате и снимает установку флага ios::scientific;

б) ios::scientific. Установка данного флага вызывает вывод чисел с плавающей запятой в научном (экспоненциальном) формате и снимает установку флага ios::fixed;

в) ios::showpoint. Установка этого флага вызывает обязательный вывод десятичного разделителя и завершающих нулей;

г) ios::showpos. Установка данного флага вызывает вывод перед положительными числами знака плюс;

д) ios::right. Установка этого флага вызывает вывод следующего элемента с правого края поля, ширина которого задана посредством функции-члена width. Для этого перед элементом выводится некоторое количество пробелов, дополняющих его до заданной ширины. Установка флага снимает установку флага ios::left;

е) ios::left. Установка флага вызывает вывод следующего элемента с левого края поля, ширина которого задана посредством функции-члена width. Для

этого после элемента выводится некоторое количество пробелов, дополняющих его до заданной ширины. Установка флага снимает установку флага ios::right.

18. Для этого поток outstream нужно заменить потоком cout и удалить объявление потока, а также вызовы функций open и close, поскольку поток cout не нуждается в объявлении, открытии и закрытии. Директива #include <fstream> включает в программу все необходимые для вывода на экран элементы класса iostream, хотя для наглядности ее можно дополнить директивой #include <iostream>.

19.
```
1
2
3
3
```

20.
```
void to_screen (ifstream& file_stream)
{
    int next;
    while (file_stream >> next)
        cout << next << endl;
}
```

21. Максимальное количество символов, которое можно присвоить строковой переменной, на единицу меньше ее объявленного размера. В данном случае оно равно 20.

22. Оператор

```
cin >> c;
```

считывает следующий символ, не являющийся пропуском, а оператор

```
cin.get(c);
```

считывает следующий символ независимо от того, является он пропуском или нет.

23. Эти два оператора эквивалентны. Оба они выводят значение переменной c.

24. Символ, помещаемый во входной поток функцией-членом putback, не обязательно должен быть последним прочитанным символом. Если ваша программа прочитала из входного потока символ a, она может записать в него с помощью функции putback любой другой символ, в том числе и b. (При этом текст во входном файле не изменится, хотя программа будет считать, что это произошло.)

25. Все данные, выводимые на экран:

```
Enter a line of input:
a b c d e f g
a b END OF OUTPUT
```

26. Все данные, выводимые на экран:

```
Enter a line of input:
abcdef gh
ace h
```

Программа выводит каждый нечетный символ входных данных, и при этом пробел считается таким же символом, как любой другой.

27. Все данные, выводимые на экран:

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
01234567891 1
```

Заметьте, что из входного значения 10 выводится только символ 1. Так происходит потому, что функция cin.get считывает не числа, а символы, и значение 10 считывается как два символа, 1 и 0. Поскольку программа выводит только нечетные символы, 0 выведен не будет. Далее выводится пробел, затем программа пропускает первую единицу числа 11, вводимого как два символа 1, и выводит вторую единицу.

28. Этот код содержит бесконечный цикл, который повторяется до тех пор, пока пользователь осуществляет ввод. Логическое выражение (next != '\n') всегда истинно, поскольку переменная next заполняется с помощью оператора

```
cin >> next;
```

который пропускает символы перевода строки '\n' (так же, как и символы пробела). Код будет выполняться, а когда пользователь закончит ввод, отобразятся следующие данные:

```
Enter a line of input:
0 1 2 3 4 5 6 7 8 9 10 11
0246811
```

Обратите внимание, что в программе из упражнения 27 для ввода используется функция cin.get, считывающая каждый символ независимо от того, является ли он пробелом; таким образом, эта программа выводит каждый нечетный символ, включая пробелы. Программа, приведенная в данном упражнении, выполняет ввод иначе — она пользуется оператором ввода >>, пропускающим символы пробела и учитывающим только значащие символы (в данном случае цифры от 0 до 9). Эта программа выводит нечетные символы, не являющиеся пропусками, и две выведенные ею единицы в конце строки представляют собой первые символы чисел 10 и 11.

29. Этот вызов функции вернет значение false. Для того чтобы он вернул true, программа сначала должна попытаться прочитать из входного потока еще один символ (после прочтения последнего символа файла).

30.
```
void text_to_screen(ifstream& file_stream)
{
    char next;
    file_stream.get(next);
    while (! file_stream.eof())
    {
        cout << next;
        file_stream.get(next);
    }
}
```

Оператор cout << next при желании можно заменить функцией cout.put(next).

31. Все данные, выводимые на экран:

```
Enter a line of input:
I'll see you at 10:30 AM.
I'll see you at 1<END OF OUTPUT
```

32.
```
cout << "Enter a line of input:\n";
char next;
do
{
    cin.get(next);
    if (!isupper(next))
        cout << next;
} while (next != '\n');
```

Обратите внимание, что в данном случае следует использовать выражение `!isupper(next)`, а не `islower(next)`, поскольку второе выражение возвращает значение `false`, когда переменная `next` содержит символ, не являющийся буквой (например, пробел или запятую).

33. Объект `cin` имеет тип `istream`, а объект `cout` — тип `ostream`.

34.
```
void copy_char(istream& source_file)
{
    char next;
    source_file.get(next);
    cout << next;
}
```

35.
```
void copy_line(istream& source_file)
{
    char next;
    do
    {
        source_file.get(next);
        cout << next;
    } while (next != '\n');
}
```

36.
```
void send_line(ostream& target_stream)
{
    char next;
    do
    {
        cin.get(next);
        target_stream << next;
    } while (next != '\n');
}
```

37.
```
2.0 1.1 2.3
2.0 3.0 2.3
2.0 3.0 4.0
```

38. Один набор функций таков:

```
void func(double x)
{
    double y = 1.1;
    double z = 2.3;
```

```
    cout << x << " " << y <<  " " << z << endl;
}
void func(double x, double y)
{
    double z = 2.3;
    cout << x << " " << y <<  " " << z << endl;
}
void func(double x, double y, double z)
{
    cout << x << " " << y <<  " " << z << endl;
}
```

Практические задания

1. Напишите программу, считывающую из файла список чисел типа `int` и выводящую на экран наименьшее и наибольшее из них. Кроме чисел, разделенных пробелами и символами разрыва строк, в файле нет никаких других данных.
2. Напишите программу, считывающую из файла список чисел типа `double` и выводящую на экран их среднее арифметическое. Помимо чисел, разделенных пробелами и символами разрыва строк, в файле нет никаких других данных.
3. Напишите программу, принимающую входные данные из файла, содержащего список чисел типа `double`. Программа выводит на экран среднеквадратическое отклонение этих чисел. Помимо чисел, разделенных пробелами и символами разрыва строк, в файле нет никаких других данных. Среднеквадратическое отклонение списка чисел определяется как квадратный корень среднего арифметического следующих значений:

   $(n_1 - a)^2$, $(n_2 - a)^2$, $(n_3 - a)^2$ и т. д.

   здесь a — среднее арифметическое всех чисел: n_1, n_2, n_3 и т. д.

   Подсказка. Программа должна сначала прочитать весь файл и вычислить среднее арифметическое содержащихся в нем чисел, а затем закрыть его, открыть снова и вычислить среднеквадратическое отклонение.
4. Напишите программу, дающую и принимающую советы по программированию. Программа начинает свою работу с того, что выводит на экран совет и просит пользователя ввести другой совет, на этом ее работа завершается. Следующий пользователь, запустивший программу, получает совет, данный предыдущим. Совет сохраняется в файле, содержимое которого меняется после каждого запуска программы. Для ввода самого первого совета и сохранения его в файле можно воспользоваться текстовым редактором. Программа должна позволять пользователю вводить советы, содержащие текст любой длины с любым количеством строк. Об окончании ввода пользователь сигнализирует двумя последовательными нажатиями клавиши Enter. А программа принимает ввод и проверяет, не встречаются ли во входном потоке два подряд идущих символа `'\n'`, указывающих на конец текста.
5. Напишите программу, считывающую текст из одного файла и записывающую его отредактированную версию в другой. Редакция заключается в том, что из

текста удаляются все лишние пробелы путем замены каждой последовательности из двух или более пробелов одним. В программе должна быть определена функция, получающая в качестве аргументов входной и выходной файловые потоки.

6. Напишите программу, объединяющую списки чисел, которые хранятся в двух файлах, и записывающую результирующий список в третий файл. Каждый файл входных данных содержит список чисел типа `int`, отсортированный по возрастанию значений. В программе должна быть определена функция, получающая в качестве аргументов два входных файловых потока и один выходной.

7. Напишите программу, генерирующую персонализированную почту типа рассылки. Она принимает ввод данных и из файла, и с клавиатуры. Входной файл содержит текст письма, в котором вместо имени адресата стоят три символа #N#. Программа запрашивает у пользователя имя получателя и записывает письмо во второй файл, предварительно заменив символы #N# именем адресата. Эта комбинация символов встречаются в письме только один раз.

   Совет. Пусть программа считывает символы из входного файла, пока не встретит последовательность #N#, и каждый прочитанный символ тут же копирует в выходной файл. Встретив эту комбинацию символов, она выводит на экран запрос имени адресата. Дальше ее действия очевидны. В программе должна быть определена функция, получающая в качестве аргументов входной и выходной файловые потоки.

   Более сложная версия программы с использованием материала факультативного раздела «Использование имен файлов в качестве входных данных»: последовательность #N# может встречаться в файле произвольное количество раз. Имя адресата включает два слова — имя и фамилию, других составляющих, таких как инициалы, отчество и т. п., оно не имеет и записывается в две переменные.

8. Напишите программу, вычисляющую итоговые оценки студентов за учебный курс. Записи с текущими оценками находятся в файле, имеющем следующий формат. Каждая строка содержит фамилию студента, один пробел, имя, еще один пробел и затем десять оценок за тесты. Оценки представляют собой целые числа, разделенные пробелами. Программа считывает входные данные из этого файла и записывает выходные данные в другой файл. Данные выходного файла совпадают с данными входного с той разницей, что в конце каждой строки стоит дополнительное число типа `double`, являющееся средним арифметическим десяти оценок студента. В программе должна быть использована хотя бы одна функция, принимающая в качестве аргументов файловые потоки (а возможно, и другие данные).

9. Дополните программу, реализующую задание 8, следующими возможностями:

   а) список оценок за тесты в каждой строке может содержать менее десяти оценок. (Если их меньше десяти, значит, студент пропустил один или более тестов.) Однако средняя оценка все равно вычисляется как общая сумма баллов, деленная на 10, как если бы студент получил оценку 0 за каждый пропущенный тест;

б) в начале файла, куда выводится вся информация, должна располагаться одна или несколько строк с пояснениями содержащихся в нем данных. Чтобы текст выглядел аккуратно и легко читался, воспользуйтесь командами форматирования;

в) после помещения выходных данных в файл программа закрывает все файлы и копирует содержимое выходного файла во входной, так что результатом ее выполнения является изменение содержимого входного файла. В программе должны быть использованы хотя бы две функции, все или некоторые аргументы которых представляют собой файловые потоки.

10. Напишите программу, вычисляющую среднюю длину слова (среднее количество символов в нем) для файла, содержащего некоторый текст. Слово определяется как строка символов, которой предшествует и за которой следует один из перечисленных элементов: пробел, запятая, точка, символ начала или конца строки. В программе необходимо определить функцию, получающую в качестве аргумента входной файловый поток. Эта функция должна работать и с потоком `cin`, хотя в качестве аргумента он ей в данной программе передаваться не будет.

11. Напишите программу, которая исправляет программу на C++, содержащую ошибки в операторах ввода и вывода. Автор программы постоянно путает, какой из операторов, >> или <<, должен использоваться с каждым из двух предопределенных потоков, `cin` и `cout`. Она заменяет неправильные вхождения:

    ```
    cin <<
    ```

    правильным вариантом:

    ```
    cin >>
    ```

    и неправильные вхождения:

    ```
    cout >>
    ```

    правильным вариантом:

    ```
    cout <<
    ```

    В упрощенной версии программы считается, что между объектом `cin` и оператором <<, а также между объектом `cout` и оператором >> всегда располагается строго один пробел.

    В более сложной версии программы допускается, что между `cin` и <<, а также между `cout` и >> может быть более одного пробела или он отсутствует вообще. Исходная программа находится в файле, а ее исправленная версия записывается в другой файл. В программе должна быть определена функция, получающая в качестве аргументов входной и выходной файловые потоки.

    Совет. Даже при создании более сложной версии дело упростится, если сначала написать простую программу, а затем ее доработать.

12. Напишите программу, позволяющую пользователю ввести однострочный вопрос и отвечающую на этот вопрос. (На самом деле программа будет считывать

вопрос пользователя и полностью его игнорировать.) Она всегда выводит один из следующих ответов:

```
I'm not sure, but I think you will find the answer in chapter #N.
That's a good question.
If I were you, I would not worry about such things.
That question has puzzled philosophers for centuries.
I don't know. I'm just a machine.
Think about it and the answer will come to you.
I used to know the answer to that question, but I've forgotten it.
The answer can be found in a secret place in the woods.
```

Эти ответы хранятся в файле (по одному в строке), и программа просто считывает из файла очередную строку, а затем выводит ее в ответ на вопрос. Прочитав весь файл, она закрывает его, открывает снова и начинает читать ответы сначала.

Выдавая первый ответ, программа должна заменить комбинацию символов #N числом от 1 до 17. Для выбора числа она может инициализировать переменную значением 17 и уменьшать ее значение на единицу каждый раз, когда нужно вывести очередное число, так что отсчет будет вестись от 17 до 1. Когда переменная достигнет значения 0, программа должна снова присвоить ей значение 17. Определите в программе с помощью квалификатора const глобальную именованную константу NUMBER_OF_CHAPTERS и присвойте ей значение 17.

Совет. Воспользуйтесь функцией new_line, рассмотренной в этой главе.

13. Этот проект похож на предыдущий, но в нем используется более сложный метод выбора ответа. Прочитав вопрос, программа должна подсчитать количество содержащихся в нем символов и сохранить его в переменной count, а затем выдать ответ под номером count%ANSWERS. Первый ответ в файле имеет номер 0, второй — 1, третий — 2 и т. д. Константа ANSWERS объявлена как глобальная именованная константа, и ее значением является количество ответов в файле:

```
const int ANSWERS = 8;
```

Это позволяет модифицировать файл ответов, добавляя их или удаляя, при этом программе достаточно изменять только объявление константы. Лишь первый ответ в файле должен всегда оставаться одним и тем же:

```
I'm not sure, but I think you will find the answer in chapter #N.
```

Заменяя комбинацию символов #N номером главы, используйте значение

```
count%NUMBER_OF_CHAPTERS + 1
```

где count — описанная выше переменная, а NUMBER_OF_CHAPTERS — глобальная именованная константа, равная количеству глав в книге.

14. Напишите программу, нумерующую строки текстового файла. Она должна считывать текст из файла и выводить каждую строку на экран и в другой файл, предварив ее номером строки. Номер выводится в поле шириной три символа в начале строки, выровненной по правому краю. За ним следует двоеточие, один пробел и текст строки. Программа должна считывать строки по

одному символу и игнорировать ведущие пробелы. Предполагается, что строки достаточно коротки и умещаются на экране. В противном случае при выводе на экран они будут переноситься на следующую строку или обрезаться в зависимости от текущих установок экрана.

Более сложная версия программы сначала определяет необходимую ширину поля номера строки, подсчитав количество строк в файле. Кроме того, она вставляет символ перевода строки после последнего полного слова, умещающегося в строку длиной 72 символа.

15. Напишите программу, которая вычисляет следующие статистические данные о файле: общее количество символов в файле, общее количество символов, не являющихся пропусками, и общее количество букв, и выводящую их на экран, а также в другой файл.

Глава 6 Определение классов

И молвил Морж: «Пришла пора подумать о делах:
О башмаках и сургуче, капусте, королях ...»

Льюис Кэрролл

В главе 5 рассказывалось о том, как пользоваться классами и объектами, но ничего не говорилось о способах их определения. Класс — это тип данных. Определяемые программистом классы применяются точно так же, как и предопределенные типы данных `int`, `char`, `ifstream` и т. п. Однако чтобы обеспечить это, классы следует определять надлежащим образом. Поэтому вам необходимо научиться правильно определять классы и овладеть некоторыми методиками, помогающими писать такие определения в соответствии с современными принципами программирования.

Прежде чем рассматривать классы, мы познакомим вас со *структурами*. Поскольку структуры, которые используются так, как рассказывается в этой главе, напоминают упрощенные классы, с их изучения можно начать освоение классов.

6.1. Структуры

Как было сказано в главе 5, объект — это переменная, включающая функции-члены, а класс — тип данных, переменные которого являются объектами. Поэтому определение класса должно содержать информацию о том, какие типы значений могут храниться в переменной и какие она имеет функции-члены. Сначала вы узнаете, как определяется тип структуры — подобия объекта, не имеющего функций-членов. А после знакомства со структурами мы перейдем непосредственно к изучению классов.

Структуры для разнородных данных

Иногда полезно иметь набор значений разных типов и оперировать им как одним элементом. В качестве примера рассмотрим банковский депозитный сертификат,

часто называемый CD (Certificate of Deposit). Это банковский счет, деньги с которого нельзя снимать в течение заданного количества месяцев. С ним связаны три элемента данных: баланс на счету, процентная ставка и значение, соответствующее сроку, по истечении которого разрешается снимать деньги. Первые два элемента могут быть представлены значениями типа double, а количество месяцев — значением типа int. В листинге 6.1 приведено определение структуры с именем CDAccount, подходящей для хранения такой информации. Это определение входит в состав демонстрационной программы. Предположим, что данный банк специализируется на краткосрочных CD, так что срок замораживания вклада не может превышать 12 месяцев. Давайте посмотрим, как определяется и используется эта структура. Ее определение следующее:

```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term;   // Количество месяцев до наступления срока выплаты.
};
```

Ключевое слово struct указывает, что это определение типа структуры. Идентификатор CDAccount является именем типа этой структуры или *тегом*. Тегом структуры может быть любой допустимый идентификатор, но не ключевое слово. Хотя язык C++ этого не требует, в тегах структур обычно используются строчные и заглавные буквы, причем начинается тег с заглавной буквы. Идентификаторы, объявленные внутри фигурных скобок, называются членами структуры. Как видно из примера, определение структуры завершается закрывающей фигурной скобкой и точкой с запятой.

Определение типа структуры обычно размещается вне определений функций (подобно определениям глобальных именованных констант). Тогда этот тип становится доступным для всего последующего кода.

Листинг 6.1. Определение структуры

```
// Программа, демонстрирующая использование типа структуры CDAccount.
#include <iostream>
using namespace std;

// Структура для депозитного банковского сертификата.
struct CDAccount
{
    double balance;
    double interest_rate;
    int term;   // Количество месяцев до наступления срока выплаты.
};

void get_data(CDAccount& the_account);
// Постусловие: переменным the_account.balance
// и the_account.interest_rate присвоены значения,
// введенные пользователем с клавиатуры.

int main()
{
    CDAccount account;
```

продолжение

Листинг 6.1 (*продолжение*)

```
    get_data(account);

    double rate_fraction, interest;
    rate_fraction = account.interest_rate/100.0;
    interest = account.balance*rate_fraction*(account.term/12.0);
    account.balance = account.balance + interest;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "When your CD matures in "
         << account.term << " months,\n"
         << "it will have a balance of $"
         << account.balance << endl;
    return 0;
}

// Используем библиотеку классов iostream.
void get_data(CDAccount& the_account)
{
    cout << "Enter account balance: $";
    cin >> the_account.balance;
    cout << "Enter account interest rate: ";
    cin >> the_account.interest_rate;
    cout << "Enter the number of months until maturity\n"
         << "(must be 12 or fewer months): ";
    cin >> the_account.term;
}
```

Пример диалога

```
Enter account balance: $100.00
Enter account interest rate: 10.0
Enter the number of months until maturity
(must be 12 or fewer months): 6
When your CD matures in 6 months,
it will have a balance of $105.00
```

После определения структуры ее тип может использоваться так же, как любой из предопределенных типов `int`, `char` и т. п. Например, следующий оператор объявляет две переменные типа `CDAccount`: `my_account` и `your_account`.

```
CDAccount my_account, your_account;
```

В переменной типа структуры, как и в любой другой переменной, могут храниться значения. Значением такой переменной является набор меньших значений, называемых значениями-членами. Каждому имени члена структуры соответствует одно значение. В частности, значение типа `CDAccount` включает три значения-члена: два типа `double` и одно типа `int`. Значения-члены хранятся в переменных-членах, о которых рассказывается далее.

В каждой структуре определяется список имен ее членов. Так, структура `CDAccount` (листинг 6.1) имеет три члена: `balance`, `interest_rate` и `term`. Каждое из имен может использоваться для доступа к одной из переменных, входящих в состав структуры. Эти переменные называются *переменными-членами* структуры. Для указания

переменной-члена нужно задать имя переменной типа структуры, точку и имя члена структуры. Например, если переменная account имеет тип CDAccount, она содержит три члена:

```
account.balance
account.interest_rate
account.term
```

Первые два имеют тип double, а третий — int. В программе они используются так же, как и переменные любых других типов. Приведенным выше переменным-членам можно присвоить значения с помощью следующих операторов присваивания:

```
account.balance = 1000.00;
account.interest_rate = 4.7;
account.term = 11;
```

Результаты выполнения этих операторов показаны на рис. 6.1. Переменные-члены структур могут использоваться во всех операциях, в которых применяются обычные переменные. Так, следующая строка программы, приведенной в листинге 6.1:

```
account.balance = account.balance + interest;
```

складывает значение переменной-члена account.balance со значением обычной переменной interest и присваивает результат переменной-члену account.balance

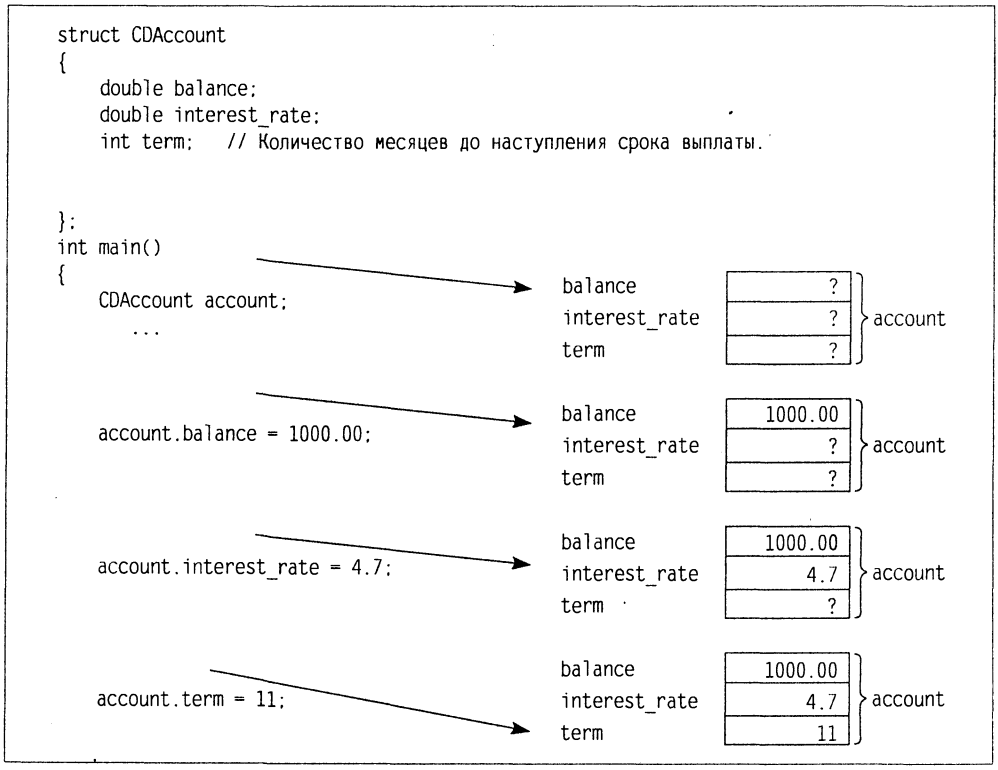


Рис. 6.1. Члены структуры

Обратите внимание, что переменная-член структуры задается с помощью оператора точка (.) так же, как члены классов, рассмотренные в главе 5. Только члены класса, с которыми мы работали в главе 5, были функциями, а члены структуры являются переменными.

Одни и те же имена переменных-членов могут использоваться в структурах разных типов. К примеру, в одной программе могут быть объявлены два следующих типа структур:

```
struct FertilizerStock
{
    double quantity; // Количество внесенных удобрений.
    double nitrogen_content;
};
```

и

```
struct CropYield
{
    int quantity;   // Собранный урожай яблок.
    double size;
};
```

Так как доступ к членам структуры всегда осуществляется по имени этой структуры, совпадение имен членов разных структур не представляет проблемы. Предположим, вы объявили следующие две переменные структурных типов:

```
FertilizerStock super_grow;
CropYield apples;
```

Тогда значение, соответствующее количеству внесенных удобрений, будет храниться в переменной `super_grow.quantity`, а количество собранных яблок — в переменной `apples.quantity`. Переменная структуры, указанная перед оператором точка (.), определяет, какая именно переменная `quantity` имеется в виду в каждом конкретном случае.

Значение структуры можно рассматривать как набор значений ее членов или же как одно комплексное значение. Второе представление позволяет использовать структурные переменные и их значения так же, как простые переменные предопределенных типов, скажем `int`. В частности, им можно присваивать значения с помощью оператора присваивания. Например, если переменные `apples` и `oranges` являются структурами типа `CropYield`, допустимо следующее присваивание:

```
apples = oranges;
```

Оно эквивалентно паре операторов:

```
apples.quantity = oranges.quantity;
apples.size = oranges.size;
```

Ловушка: отсутствие точки с запятой в определении структуры

Закрывающей фигурной скобки в определении структуры не достаточно для завершения ее определения — необходимо еще поставить точку с запятой. Вы можете

спросить, почему, скажем, определение функции завершается только фигурной скобкой, а в определении структуры нужна еще и точка с запятой. Дело в том, что определение структуры – это не просто определение. Оно может использоваться и для одновременного объявления переменных типа данной структуры: в таком случае список переменных располагается между фигурной скобкой и завершающей точкой с запятой. В частности, следующий фрагмент кода определяет структуру WeatherData и две переменные типа этой структуры, data_point1 и data_point2:

```
struct WeatherData
{
    double temperature;
    double wind_velocity;
} data_point1, data_point2;
```

Таким образом, полный синтаксис объявления структуры (ключевое слово struct) предполагает обязательное определение ее типа и возможное наличие объявлений переменных, а также требует использования точки с запятой, чтобы отделить код объявления этой структуры от других объявлений и операторов программы.

Оператор точка (.)

Оператор точка (.) используется для доступа к члену структуры.

Синтаксис

имя_переменной_типа_структуры.имя_члена_структуры

Примеры

```
struct StudentRecord
{
    int student_number;
    char grade;
};

int main()
{
    StudentRecord your_record;
    your_record.student_number = 2001;
    your_record.grade = 'A';
```

В контексте структур оператор точка (.) называется также *оператором доступа к члену структуры.*

Структуры как аргументы функций

Функции могут иметь параметры структурных типов, передаваемые по значению или по ссылке. Например, программа в листинге 6.1 включает функцию get_data, параметр которой имеет тип структуры CDAccount и передается по ссылке.

Возвращаемое функцией значение тоже может иметь тип структуры. В частности, следующая функция принимает три простых аргумента и возвращает составное значение типа CDAccount:

```
CDAccount shrink_wrap(double the_balance,
                      double the_rate, int the_term)
```

```
{
    CDAccount temp;
    temp.balance = the_balance;
    temp.interest_rate = the_rate;
    temp.term = the_term;
    return temp;
}
```

Обратите внимание на локальную переменную temp типа CDAccount. Она используется для формирования полного значения типа структуры, возвращаемого функцией. После определения функции shrink_wrap с ее помощью можно присваивать значения переменным типа CDAccount:

```
CDAccount new_account;
new_account = shrink_wrap(10000.00, 5.1, 11);
```

Совет программисту: пользуйтесь иерархическими структурами

В некоторых случаях данные программы удобно представить в виде структуры, некоторые члены которой сами являются структурами. Так, структуру типа PersonInfo, предназначенную для хранения информации о росте, весе и дате рождения человека, можно определить следующим образом:

```
struct Date
{
    int month;  // Месяц
    int day;    // День
    int year;   // Год
};
struct PersonInfo
{
    double height;   // Рост в дюймах
    int weight;      // Вес в фунтах
    Date birthday;   // Дата рождения
};
```

Переменная типа PersonInfo объявляется обычным образом:

```
PersonInfo person1;
```

Если в переменную person1 записана дата рождения человека, то год его рождения можно вывести на экран так:

```
cout << person1.birthday.year;
```

Подобные выражения читаются слева направо. Здесь person1 — переменная типа структуры PersonInfo. Для получения переменной-члена birthday используется оператор точка (.):

```
person1.birthday
```

Эта переменная, являющаяся членом структуры типа PersonInfo, сама представляет собой тип структуры Date. Ее переменная-член year также извлекается с помощью названного оператора, и в результате получается выражение person1.birthday.year.

Инициализация структур

Структуру можно инициализировать при ее объявлении. Для того чтобы присвоить значение переменной типа структуры, нужно поместить после ее имени знак равенства и в фигурных скобках задать значения всех членов этой структуры. Рассмотрим определение типа структуры для хранения даты рождения, приведенное в предыдущем разделе:

```
struct Date
{
    int month;    // Месяц
    int day;      // День
    int year;     // Год
};
```

После определения типа Date можно следующим образом объявить и инициализировать переменную due_date этого типа:

```
Date due_date ={12, 31, 2004};
```

Обратите внимание, что значения членов структуры должны задаваться в точности в том же порядке, в каком их имена следуют в определении типа структуры. В этом примере переменная-член due_date.month инициализируется значением 12, переменная-член due_date.day — значением 31, а переменная-член due_date.year — значением 2004.

Число значений, заданных при инициализации структуры, не может превышать количество ее членов, но может быть меньше. В последнем случае заданные значения используются для инициализации членов структуры, начиная с первого и далее по порядку, а те члены, для которых значений не хватило, инициализируются нулевыми значениями соответствующего типа.

Простые типы структур

Именем типа структуры является *тег*.

Синтаксис

```
struct тег
{
    тип_1 имя_переменной_члена_1;
    тип_2 имя_переменной_члена_2;

            ...
    тип_n имя_переменной_члена_n;
};   // Не забудьте эту точку с запятой.
```

Пример

```
struct Automobile
{
    int year;
    int doors;
    double horse_power;
    char model;
};
```

продолжение

Имена членов структуры одного типа можно задавать в одной строке кода через запятую, хотя в этой книге мы этого делать не будем. Так, следующее определение:

```
struct Automobile
{
int year, doors;
double horse_power;
    char model;
};
```

эквивалентно предыдущему.

Переменные типа структуры объявляются аналогично переменным других типов, например:

```
Automobile my_car, your_car;
```

Переменные-члены задаются с помощью оператора точка (.): `my_car.year`, `my_car.doors`, `my_car.horse_power` или `my_car.model`.

Упражнения для самопроверки

1. Допустим, есть следующее определение типа структуры и объявление переменной:

```
struct TermAccount
{
    double balance;
    double interest_rate;
    int term;
    char initial1;
    char initial2;
};

TermAccount account;
```

Укажите тип каждого из перечисленных ниже элементов:

а) `account.balance;`

б) `account.interest_rate;`

в) `TermAccount.term;`

г) `savings_account.initial1;`

д) `account.initial2;`

е) `account.`

2. В программе определен такой тип структуры:

```
struct ShoeType
{
    char style;
    double price;
};
```

Что выведет приведенный ниже код из этой программы:

```
ShoeType shoe1, shoe2;
shoe1.style = 'A';
shoe1.price = 9.99;
```

```
cout << shoe1.style << " $" << shoe1.price << endl;
shoe2 = shoe1;
shoe2.price = shoe2.price/9;
cout << shoe2.style << " $" << shoe2.price << endl;
```

3. Какую ошибку содержит следующее определение структуры:

```
struct Stuff
{
    int b;
    int c;
}
int main()
{
    Stuff x;
    // Остальной код
}
```

Какое сообщение выведет для нее компилятор?

4. Имеется следующее определение структуры:

```
struct A
{
    int member_b;
    int member_c;
};
```

Объявите переменную x типа этой структуры. Инициализируйте члены member_b и member_c структуры x значениями 1 и 2 соответственно.

Примечание. В данном упражнении нужно именно инициализировать переменные-члены, а не присвоить им значения. Это существенное различие, и далее в этой главе мы к нему еще вернемся.

5. Ниже приведены примеры инициализации переменной типа структуры.

```
struct Date
{
    int month;
    int day;
    int year;
};
```

а) `Date due_date = {12, 21};`

б) `Date due_date = {12, 21, 2022};`

в) `Date due_date = {12, 21, 20, 22};`

г) `Date due_date = {12, 21, 22};`

Что происходит в каждом из четырех случаев? Если что-то не так, укажите, что именно.

6. Напишите определение типа структуры для записей, содержащих ставку сотрудника, срок предоставленного ему отпуска (целое количество дней) и вид оплаты (фиксированный оклад или почасовая оплата). Вид оплаты обозначается одним из двух значений типа char: 'H' и 'S'. Назовите тип EmployeeRecord.

7. Приведите определение функции, соответствующее следующему объявлению. (Объявление типа ShoeType приведено в упражнении 2.)

```
void read_shoe_record(ShoeType& new_shoe);
// Заполняет new_shoe значениями, введенными с клавиатуры.
```

8. Приведите определение функции, соответствующее следующему объявлению. (Объявление типа ShoeType приведено в упражнении 2.)

```
ShoeType discount(ShoeType old_record);
// Возвращает структуру, полученную в качестве аргумента,
// но с ценой (price), сниженной на 10 %.
```

9. Приведите определение структуры StockRecord с двумя переменными-членами: shoe_info типа ShoeType, объявленной в упражнении 2, и arrival_date типа Date, объявленной в упражнении 5.
10. Объявите переменную типа StockRecord (см. предыдущее упражнение) и напишите оператор, устанавливающий год (year) даты прибытия (arrival_date), равным 2004.

6.2. Классы

Я не хочу быть членом ни одного клуба,
в который меня готовы принять.

Гручо Маркс

Определение классов и функций-членов

Класс — это тип данных, переменные которого являются объектами. В главе 5 объект рассматривался как переменная, содержащая не только значения данных[1], но и функции-члены. Таким образом, в программе, созданной на C++, определение класса должно быть определением типа данных, содержащим информацию о значениях, которые могут храниться в его переменных, а также о функциях-членах этого класса. Определение структуры содержит только часть указанной информации. Структура — это определяемый программистом составной тип данных, содержащий переменные-члены. Для того чтобы структуру превратить в класс, нужно добавить в нее функции-члены.

Простейшим примером определения класса служит программа из листинга 6.2. Определенный в ней тип DayOfYear представляет собой класс объектов, значениями которых являются даты, скажем 1 января или 4 июля. Объекты этого типа могут использоваться для хранения дат праздников, дней рождения и т. п. Месяц в них задается как значение типа int: 1 соответствует январю, 2 — февралю и т. д. День месяца хранится во второй переменной-члене типа int. Класс DayOfYear включает

[1] В действительности объект является значением переменной, а не самой переменной, но поскольку мы пользуемся переменными для именования содержащихся в них значений, то не будем делать различий между переменной и ее значением.

одну функцию-член output, которая не имеет аргументов и выводит на экран значения месяца и дня. А теперь давайте рассмотрим определение класса DayOfYear подробнее.

Класс DayOfYear определен в начале программного кода, приведенного в листинге 6.2, перед функцией main. О строке, которая содержит ключевое слово public, мы пока говорить не будем. Она указывает только на то, что доступ к переменным и функциям, являющимся членами класса, не ограничен. Остальная часть определения класса DayOfYear очень похожа на определение структуры. Отличие лишь в том, что вместо ключевого слова struct используется ключевое слово class и среди членов класса наряду с переменными month и day имеется функция output. Обратите внимание, что для функции приведено только ее объявление. Определения функций-членов класса располагаются вне его объявления. (В определении класса в C++ можно чередовать переменные-члены и функции-члены, объявляя их в любом порядке, но мы придерживаемся более строгого стиля и всегда размещаем сначала список всех функций, а затем список переменных.) Объекты класса (то есть переменные типа класса) объявляются так же, как переменные предопределенных типов и типов структур.

Функции-члены определяемых программистом классов вызываются аналогично функциям-членам предопределенных классов, с которыми мы работали в главе 5. Например, в программе листинга 6.2 два объекта типа DayOfYear объявляются следующим образом:

```
DayOfYear today, birthday;
```

Функция-член output объекта today вызывается так:

```
today.output();
```

А вызов функции-члена output объекта birthday такой:

```
birthday.output();
```

Определение функции-члена должно содержать имя класса, так как могут существовать два или более классов с одинаковыми именами. В программе из листинга 6.2 определен только один класс, но в целом программа может включать определения множества классов, а функция-член output может присутствовать в каждом из них. Ее определение для класса DayOfYear приведено в листинге 6.2. Оно похоже на определение обычной функции, но имеет ряд особенностей.

Листинг 6.2. Класс с функцией-членом

```
// Программа, демонстрирующая очень простой пример класса.
// Более удачный пример класса DayOfYear будет приведен в листинге 6.3.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    void output();   // Объявление функции-члена.
    int month;
    int day;
};
```

продолжение

Листинг 6.2 (*продолжение*)

```
int main()
{
    DayOfYear today, birthday;

    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;

    cout << "Today's date is: ";
today.output();   // Вызов функции-члена.
    cout << "Your birthday is: ";
birthday.output();   // Вызов функции-члена.

    if (today.month == birthday.month
        && today.day == birthday.day)
        cout << "Happy Birthday!\n";
    else
        cout << "Happy Unbirthday!\n";

    return 0;
}

// Используем библиотеку классов iostream.
void DayOfYear::output()   // Определение функции-члена.
{
    cout << "month = " << month
         << ". day = " << day << endl;
}
```

Пример диалога

```
Enter today's date:
Enter month as a number: 10
Enter the day of the month: 15
Enter your birthday:
Enter month as a number: 2
Enter the day of the month: 21
TodayTs date is month = 10. day = 15
Your birthday is month = 2. day = 21
Happy Unbirthday!
```

Инкапсуляция

Объединение группы элементов программы, в частности переменных или функций, в одной программной структуре, например такой, как объект некоторого класса, называется *инкапсуляцией.*

Заголовок определения функции-члена output класса DayOfYear имеет вид:

```
void DayOfYear::output()
```

Оператор :: называется *оператором доступа к члену класса по имени класса*, а его назначение подобно назначению оператора точка (.). Оба они указывают, членом какого класса является данная функция. Но если первый оператор используется с именем класса, то второй – с именем объекта (то есть переменной класса). Оператор :: состоит из двух двоеточий без пробела между ними. Предшествующее ему имя класса часто называют уточнителем типа, поскольку оно определяет («уточняет»), к какому типу относится конкретное имя.

Посмотрите на определение функции-члена DayOfYear::output в листинге 6.2. Обратите внимание, что здесь используются имена членов класса month и day, причем сами по себе без указания объекта и оператора точка (.). И в этом нет ничего удивительного. Определение функции output относится ко всем объектам типа DayOfYear, и имена конкретных объектов, для которых она будет вызываться, нам не известны. Но когда эта функция будет вызвана, как в следующем операторе:

```
today.output();
```

все имена членов класса, используемые в определении функции, будут «уточнены» именем вызывающего объекта. Поэтому приведенный выше вызов эквивалентен выполнению следующего кода:

```
{
    cout << "month = " << today.month
         << ", day = " << today.day << endl;
}
```

В определении функции-члена класса можно использовать имена всех членов этого класса (как переменных, так и функций) без оператора точка (.).

Упражнения для самопроверки

11. Ниже приводится модифицированное определение класса DayOfYear из программы листинга 6.2: в него добавлена еще одна функция-член с именем input. Напишите определение этой функции.

```
class DayOfYear
{
public:
    void input();
    void output();
    int month;
    int day;
};
```

12. Имеется следующее определение класса. Напишите соответствующее определение функции-члена set:

```
class Temperature
{
public:
    void set(double new_degrees, char new_scale);
    // Присваивает переменным-членам значения, заданные в качестве аргументов.
```

```
    double degrees;
    char scale;   // 'F' – для значений температуры по Фаренгейту
                  // и 'C' – для значений температуры по Цельсию.
};
```

13. Опишите различие между оператором точка и оператором доступа к члену класса по имени класса.

Определение функции-члена

Функция-член определяется подобно любой другой функции, с той разницей, что в ее заголовке перед именем функции стоит *имя_класса* и оператор ::.

Синтаксис

```
возвращаемый_тип имя_класса::имя_функции(список_параметров)
{
тело_функции
}
```

Пример

```
// Используем библиотеку классов iostream:
void DayOfYear::output()
{
    cout << "месяц = " << month
         << ", число = " << day << endl;
}
```

Определение класса DayOfYear приведено в листинге 6.2, где month и day определены как переменные-члены этого класса. Обратите внимание, что в теле функции имена указанных переменных не предваряются именем объекта и точкой.

Оператор точка (.) и оператор ::

И оператор точка (.), и оператор :: используются с именами членов для указания класса или структуры, к которым эти члены принадлежат. Предположим, что в программе определен класс DayOfYear и объявлен объект этого класса today:

```
DayOfYear today;
```

С помощью оператора точка (.) можно обращаться к членам объекта today. В частности, функция output является членом класса DayOfYear (см. листинг 6.2) и следующий ее вызов выводит на экран данные, хранящиеся в объекте today:

```
today.output();
```

Оператор доступа к члену класса по его имени :: служит для указания имени класса в определении функции-члена этого класса. Вот, например, заголовок определения функции-члена output класса DayOfYear:

```
void DayOfYear::output()
```

Помните, что оператор :: используется с именем класса, тогда как оператор точка (.), называемый оператором непосредственного доступа к члену класса, — с объектами этого класса.

Открытые и закрытые члены класса

Предопределенные типы вроде `double` реализованы не так, как классы C++: разработчики компилятора этого языка нашли другие способы представления значений таких типов. Например, тип `double` можно реализовать множеством разных способов. Собственно говоря, в различных версиях C++ он и реализован по-разному, но предполагается, что перемещение программы, в которой используется такой тип данных, с одного компьютера на другой никак не скажется на ее работе[1]. Классы — это определяемые программистом типы, и они должны вести себя так же, как предопределенные типы. В частности, можно поместить определение класса в отдельный файл и копировать его в каждую программу, где он используется.

В разрабатываемых вами классах правила использования класса должны быть отделены от его реализации, как это сделано для предопределенных типов. Если реализация класса изменится (будет модифицировано определение функции, чтобы она работала быстрее), ни одна строка программы, в которой применяется этот класс, не должна требовать внесения изменений. Чтобы это стало понятнее, рассмотрим еще одну особенность определения классов.

Давайте вернемся к определению класса `DayOfYear` (см. листинг 6.2). Тип `DayOfYear` предназначен для хранения значений, представляющих собой даты, скажем праздников или дней рождения. Эти даты задаются двумя целыми числами, определяющими день и месяц. Позднее мы могли бы для представления месяца вместо одной переменной типа `int` применять три переменные типа `char`, в которых будут храниться три буквы сокращенного названия месяца (например, `'J'`, `'a'` и `'n'` для января — January). Однако программисту, который пользуется в своих программах классом `DayOfYear`, нет необходимости знать способ представления месяца. Конечно, в случае изменения такового в классе `DayOfYear` придется менять и реализацию функции `output`, но не более того. Остальные части программы, в которых используется данный класс, модифицировать не нужно. К сожалению, программа в листинге 6.2 не соответствует этому идеалу. Так, если заменить в классе `DayOfYear` одну переменную-член `month` тремя переменными типа `char`, придется модифицировать те части программы, где имеются обращения к этой переменной, а именно: блок ввода-вывода и оператор `if...else`.

Идеальное определение класса должно быть таким, чтобы изменение деталей реализации такого класса не требовало модификации программ, в которых он используется. Для этого класс должен содержать больше функций-членов, чем в нашем первом примере, чтобы программа никогда не обращалась к переменным-членам класса непосредственно, а оперировала данными объектов только через их функции-члены. Тогда при изменении набора переменных-членов достаточно будет модифицировать определения функций-членов, а не программу, в которой используется класс. Именно таким образом переопределен класс `DayOfYear` в программе из листинга 6.3. Внимательно просмотрев ее, вы увидите, что имена переменных-членов `month` и `day` используются только в определениях функций-членов класса.

[1] На практике это, к сожалению, не всегда осуществимо, однако разработчики компиляторов стремятся к получению идеального результата, и, по крайней мере, в случае простых программ его можно добиться.

Вне этих определений нет ни одной ссылки на переменные today.month, today.day, bach_birthday.month и bach_birthday.day.

Листинг 6.3. Класс с закрытыми членами

```
// Программа, демонстрирующая класс DayOfYear.
#include <iostream>   // Это измененная версия класса DayOfYear,
                      // первое определение которого было приведено в листинге 6.2.
using namespace std;

class DayOfYear
{
public:
void input();
    void output();

    void set(int new_month, int new_day);
    // Предусловие: значения аргументов new_month и new_day составляют допустимую дату.
    // Постусловие: дата изменена в соответствии со значениями аргументов.

    int get_month();
    // Возвращает порядковый номер месяца: 1 для января, 2 для февраля и т. д.

    int get_day();
    // Возвращает день месяца.
private:
    void check_date();   // Закрытая функция-член.
// Закрытые переменные-члены.
    int month;
    int day;
};

int main()
{
    DayOfYear today, bach_birthday;
    cout << "Enter today's date:\n";
    today.input();
    cout << "Today's date is: ";
    today.output();

    bach_birthday.set(3, 21);
    cout << "J. S. Bach's birthday is: ";
    bach_birthday.output();

    if ( today.get_month() == bach_birthday.get_month() &&
              today.get_day() == bach_birthday.get_day() )
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
```

```
}
// Используем библиотеку классов iostream.
void DayOfYear::input()
{
    cout << "Enter the month as a number: ";
    cin >> month;

// Закрытые члены могут применяться только в определениях функций-членов.

    cout << "Enter the day of the month: ";
    cin >> day;
    check_date();

// В идеале функция input должна предложить пользователю
// повторить ввод даты, если указано неверное значение, но о том,
// как это правильно реализовать, рассказывается только в главе 7.

}

// Используем библиотеку классов iostream.
void DayOfYear::output()
{
    cout << "month = " << month
         << ", day = " << day << endl;
}

void DayOfYear::set(int new_month, int new_day)
{
    month = new_month;
    day = new_day;
    check_date();
}

// Функция-член check_date проверяет не все возможные способы
// неверного задания даты, но ее легко дополнить, чтобы
// проверка была исчерпывающей (см. упражнение для самопроверки 14).

void DayOfYear::check_date()
{
    if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
    {
        cout << "Illegal date. Aborting program.\n";
        exit(1);   // Функция exit завершает работу программы.
    }
}

int DayOfYear::get_month()
{
    return month;
}
```

продолжение

Листинг 6.3 (*продолжение*)

```
int DayOfYear::get_day()
{
    return day;
}
```

Пример диалога

```
Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 21
Today's date is month = 3, day = 21
J. S. Bach's birthday is month = 3, day = 21
Happy Birthday Johann Sebastian!
```

В листинге 6.3 появился новый элемент определения класса, предназначенный для защиты переменных-членов от попыток обращения к ним извне. Это ключевое слово `private`. Все члены класса, имена которых стоят после строки с таким ключевым словом, являются *закрытыми членами класса*, то есть к ним нельзя обращаться непосредственно из программы — только из определений функций, являющихся членами того же класса. Если вы попытаетесь обратиться к закрытым переменным-членам класса из главной части программы или из определения одной из функций, не входящих в данный класс, компилятор выдаст сообщение об ошибке. При включении в список членов класса ключевого слова `private` с двоеточием все следующие за ним переменные будут объявлены закрытыми переменными-членами класса, а следующие за ним функции — закрытыми функциями-членами класса.

Все переменные-члены класса `DayOfYear` (см. листинг 6.3) являются закрытыми. Закрытые переменные-члены могут использоваться лишь в определениях любых функций-членов этого же класса. Ниже приведены два присваивания, которые недопустимы в функции `main`:

```
DayOfYear today;    // С этой строкой все в порядке.
today.month = 12;   // НЕДОПУСТИМО
today.day = 25;     // НЕДОПУСТИМО
```

Кроме того, не разрешается использовать ссылки на закрытые переменные (за исключением ссылок в определении функций-членов класса `DayOfYear`). Поскольку новое определение нашего класса делает переменные `month` и `day` закрытыми, следующий код в функции `main` тоже недопустим:

```
cout << today.month;   // НЕДОПУСТИМО
cout << today.day;     // НЕДОПУСТИМО
if (today.month == 1)  // НЕДОПУСТИМО
    cout << "January";
```

После того как переменная-член класса стала закрытой, ее значение в программе нельзя изменить, а прочитать его можно, лишь обратившись к одной из функций-членов класса и никак иначе. Это жесткое ограничение вызвано необходимостью сделать код классов более надежным, понятным и облегчить его модификацию.

Может показаться, что программа из листинга 6.3 не так уж ограничивает доступ к закрытым переменным-членам класса, поскольку их значения можно изменять

и считывать с помощью функций-членов DayOfYear::set, DayOfYear::get_month и DayOfYear::get_day. Однако если изменить представление месяца и числа даты, это впечатление исчезнет. Предположим, что определение типа DayOfYear изменилось следующим образом:

```
class DayOfYear
{
public:
    void input();
    void output();

    void set(int new_month, int new_day);
    // Предусловие: значения аргументов new_month и new_day составляют допустимую дату.
    // Постусловие: дата изменена в соответствии со значениями аргументов.

    int get_month();
    // Возвращает порядковый номер месяца: 1 для января, 2 для февраля и т. д.

    int get_day();
    // Возвращает день месяца.

private:
    void DayOfYear::check_date();
    char first_letter;   // Первая буква сокращенного названия месяца.
    char second_letter;  // Вторая буква сокращенного названия месяца.
    char third_letter;   // Третья буква сокращенного названия месяца.
    int day;
};
```

Теперь функции-члены определить немного сложнее, но с точки зрения программы они должны вести себя аналогично их поведению в предыдущем определении класса. Например, определение функции get_month может начинаться так:

```
int DayOfYear::get_month()
{
    if (first_letter == 'J' && second_letter == 'a'
            && third_letter == 'n')
        return 1;
    if (first_letter == 'F' && second_letter == 'e'
            && third_letter == 'b')
        return 2;
        ...
```

Возможно, составлять его несколько утомительно, но вовсе не трудно.

Кроме того, обратите внимание, что функции-члены DayOfYear::set и DayOfYear::input перед установкой даты проверяют, допустимы ли введенные значения числа и месяца. Для этого они вызывают функцию-член DayOfYear::check_date. Если бы переменные-члены month и day были не закрытыми, а открытыми, им можно было бы присваивать любые значения, в том числе и недопустимые. Закрыв эти переменные от внешнего доступа и разрешив их установку только через соответствующие функции-члены класса, мы гарантируем, что им никогда не будут присвоены недопустимые значения. (В упражнении для самопроверки 14 вам будет предложено переопределить функцию-член DayOfYear::check_date так, чтобы она выполняла исчерпывающую проверку допустимости заданной даты.)

Функции-члены класса тоже могут быть закрытыми. Как и закрытые переменные, такие функции применяются только в определениях других функций-членов класса, то есть доступ к ним из программы, в которой используется данный класс (скажем, из функции main или любой другой функции), невозможен. Так, функция-член DayOfYear::check_date (см. листинг 6.3) является закрытой. Как правило, закрытыми объявляют те функции-члены класса, которые играют вспомогательную роль в определениях других функций-членов этого класса, и для работы с ним в программе не нужны.

Ключевое слово public служит для объявления *открытых членов класса* точно так же, как ключевое слово private — для объявления закрытых членов. Например, в классе DayOfYear все функции-члены, за исключением DayOfYear::check_date, объявлены как открытые (тогда как все переменные-члены — как закрытые). Открытые члены класса могут применяться в теле функции main и других функций программы, не входящих в этот класс.

Ключевые слова public и private можно использовать в определении класса многократно. Когда в списке членов класса встречается очередное ключевое слово public, следующие за ним (до закрывающей фигурной скобки или до следующего ключевого слова private) переменные и функции объявляются открытыми, а когда встречается очередное ключевое слово private, следующие за ним (до закрывающей фигурной скобки или до следующего ключевого слова public) переменные и функции объявляются закрытыми. Так, в приведенном ниже определении класса функция-член do_something_else и переменная-член more_stuff являются закрытыми, а все остальные функции и переменные — открытыми.

```
class SampleClass
{
public:
    void do_something();
    int stuff;
private:
    void do_something_else();
    char more_stuff;
public:
    double do_yet_another_thing();
    double even_more_stuff;
};
```

Если группе членов класса, расположенной в начале его определения, не предшествует ни одно из ключевых слов (public или private), все они считаются закрытыми. Однако лучше всегда явно определять каждую группу членов класса с помощью соответствующих ключевых слов.

Совет программисту: объявляйте все переменные-члены как закрытые

Обычно все переменные-члены класса объявляют с использованием ключевого слова private, чтобы доступ к их значениям как для записи, так и для чтения, был возможен только из функций-членов этого же класса. Данному вопросу посвящена существенная часть настоящей главы.

Классы и объекты

Класс — это тип, переменные которого являются *объектами*. Членами объектов могут быть переменные и функции. Ниже приведен синтаксис определения класса.

```
class имя_класса
{
public:
// Открытые члены
    спецификация_члена_1
    спецификация_члена_2
        ...
    спецификация_члена_n
private:
// Закрытые члены
    спецификация_члена_n+1
    спецификация_члена_n+2
        ...
};   // Не забудьте указать точку с запятой.
```

Каждая *спецификация_члена_i* представляет собой или объявление переменной-члена, или объявление функции-члена. (Допускается наличие дополнительных секций `public` и `private`.)

Рассмотрим пример.

```
class Bicycle
{
public:
    char get_color();
    int number_of_speeds();
    void set(int the_speeds, char the_color);
private:
    int speeds;
    char color;
};
```

После определения типа класса можно объявлять его объекты (то есть переменные типа класса) точно так же, как переменные других типов. Вот пример объявления двух объектов типа `Bicycle`:

```
Bicycle my_bike, your_bike;
```

Совет программисту: определяйте аксессоры и мутаторы

С помощью оператора == можно проверить, равны ли два значения одного и того же простого типа. К сожалению, стандартный предопределенный оператор == неприменим к объектам. Из главы 8 вы узнаете, как сделать, чтобы им можно было пользоваться и для сравнения объектов. А пока ни с объектами, ни со структурами вы его использовать не умеете, и это может вызывать некоторые затруднения. Определяя класс, все его переменные-члены следует делать закрытыми. Поэтому для выяснения того, содержат ли два объекта одни и те же данные, нужно каким-то образом получить доступ к их переменным-членам (или некоторому эквиваленту их значений). Сравнив попарно значения переменных-членов двух объектов,

вы узнаете, эквивалентны эти объекты или нет. В программе, приведенной в листинге 6.3, возможность такого сравнения обеспечивают функции-члены `get_month` и `get_day`. С их помощью в операторе `if...else` сравниваются два объекта типов `DayOfYear`: `today` и `bach_birthday`.

Функции-члены класса, которые, подобно функциям `get_month` и `get_day`, возвращают значения закрытых переменных объекта, называются *аксессорами*. Описанная в этой главе технология создания классов, предполагающая объявление всех переменных-членов класса закрытыми, требует обязательного создания для каждого класса полного набора аксессоров, позволяющих сравнивать объекты. Эти функции не обязательно должны возвращать значения каждой переменной-члена; их задача — предоставить набор значений, представляющих данные объекта в той или иной форме. В главе 8 описан более изящный способ сравнения двух объектов, но даже после его освоения удобно иметь в наличии аксессоры.

Функции-члены класса, позволяющие, подобно функции `set` из программы листинга 6.3, изменять значения закрытых переменных-членов класса, называются *мутаторами*. Такие функции должны входить в состав каждого класса, чтобы пользователь мог изменять данные объектов.

Аксессоры и мутаторы

Аксессоры и мутаторы являются важной составляющей каждого класса, поскольку обеспечивают возможность чтения и изменения значений объектов этого класса.

Функции-члены класса, предназначенные для получения значений закрытых переменных-членов объектов этого класса, называются *аксессорами*. Аксессор не обязательно должен возвращать реальное значение каждой переменной-члена класса — он может возвращать некоторый эквивалент данного значения. Хотя язык C++ этого не требует, обычно имена аксессоров содержат слово `get` («получить»).

Функции-члены класса, предназначенные для изменения значений закрытых переменных-членов объектов этого класса, называются *мутаторами*. Хотя зык C++ этого не требует, обычно имена мутаторов содержат слово `set` («установить»).

Упражнения для самопроверки

14. Закрытая функция-член `DayOfYear::check_date` в листинге 6.3 пропускает некоторые недопустимые даты, например 30 февраля. Переопределите эту функцию так, чтобы она завершала работу программы для любой недопустимой даты. Она должна пропускать дату 29 февраля, чтобы учесть високосные годы. (Писать такую функцию немного утомительно, поскольку она содержит длинный и однообразный код, но совсем не сложно.)
15. Предположим, что ваша программа включает такое определение класса:

```
class Automobile
{
public:
    void set_price(double new_price);
    void set_profit(double new_profit);
    double get_price();
```

```
private:
    double price;
    double profit;
    double get_profit();
};
```

Допустим также, что функция main содержит приведенное ниже объявление объектов:

```
Automobile hyundai, jaguar;
```

и программа каким-то образом инициализировала все переменные-члены этих объектов.

Какие из следующих операторов:

```
hyundai.price = 4999.99;
jaguar.set_price(30000.97);
double a_price, a_profit;
a_price = jaguar.get_price();
a_profit = jaguar.get_profit();
a_profit = hyundai.get_profit();
if (hyundai == jaguar)
cout << "Want to swap cars?";
hyundai = jaguar;
```

допустимы в функции main?

16. Предположим, что из определения класса в упражнении 15 удалена строка с ключевым словом private. Как изменится ответ этого упражнения?
17. Объясните роль ключевых слов public: и private: в определении класса. В частности, укажите, почему нельзя просто объявить все члены класса как открытые (public) и тем самым упростить доступ к данным и функциям класса.
18. Ответьте на следующие вопросы.
    а) сколько секций public: необходимо в определении класса;
    б) сколько секций private: необходимо в определении класса;
    в) к какому типу относится секция, расположенная между открывающей фигурной скобкой и первым ключевым словом public: или private: в определении класса;
    г) к какому типу относится секция, расположенная между открывающей фигурной скобкой и первым ключевым словом public: или private: в определении структуры?

Совет программисту: используйте для объектов оператор присваивания

Оператор присваивания = вполне допустимо использовать с объектами или структурами. В качестве примера предположим, что класс DayOfYear определен, как показано в листинге 6.3 (то есть содержит переменные-члены month и day), а в программе объявлены объекты due_date и tomorrow:

```
DayOfYear due_date, tomorrow;
```

В этом случае можно применять такой оператор (при условии, что переменные-члены объектов due_date и tomorrow уже имеют некоторые значения):

```
due_date = tomorrow;
```

Приведенное выше присваивание эквивалентно следующему:

```
due_date.month = tomorrow.month;
due_date.day = tomorrow.day;
```

Более того, это верно, даже если переменные month и day являются закрытыми членами класса DayOfYear[1].

Пример: класс BankAccount

В листинге 6.4 приведено определение класса для банковского счета, иллюстрирующее все уже известные вам аспекты определения классов. Данный тип счета позволяет снимать деньги в любой момент — вклад на нем не замораживается, как на счету, представленном классом CDAccount (объявлен в начале этой главы). Кроме того, класс BankAccount содержит функции-члены для выполнения всех операций, которые могут вам потребоваться. Объекты класса BankAccount включают две закрытые переменные: одну для баланса счета, а вторую для процентной ставки. Давайте рассмотрим некоторые особенности класса BankAccount.

Листинг 6.4. Класс BankAccount

```
// Программа, в которой демонстрируется использование класса BankAccount.
#include <iostream>
using namespace std;

// Класс для банковского счета:
class BankAccount
{
public:
    void set(int dollars, int cents, double rate);   // Функция-член set перегружена.
    // Постусловие: баланс счета установлен равным $dollars.cents;
    // процентная ставка задана равной значению аргумента rate.

    void set(int dollars, double rate);
    // Постусловие: баланс счета установлен равным $dollars.00;
    // процентная ставка установлена равной значению аргумента rate.

    void update();
    // Постусловие: к балансу счета прибавлены процентные
    // начисления за один год.

    double get_balance();
    // Возвращает текущий баланс счета.

    double get_rate();
    // Возвращает текущую процентную ставку.
```

[1] В главе 12 описываются ситуации, в которых оператор присваивания = должен быть переопределен (перегружен) для класса.

```
    void output(ostream& outs);
    // Предусловие: если outs – выходной файловый поток,
    // он уже соединен с файлом.
    // Постусловие: значения баланса счета и процентной ставки
    // записаны в поток outs.
private:
    double balance;
    double interest_rate;

    double fraction(double percent);
    // Преобразует проценты в дробь. Например, функция fraction(50.3)
    // возвращает значение 0.503.
};

int main()
{
    BankAccount account1, account2;
    cout << "Start of Test:\n";

    account1.set(123, 99, 3.0);   // Вызов перегруженной функции-члена set.
    cout << "account1 initial statement:\n";
    account1.output(cout);

    account1.set(100, 5.0);
    cout << "account1 with new setup:\n";
    account1.output(cout);

    account1.update();
    cout << "account1 after update:\n";
    account1.output(cout);

    account2 = account1;
    cout << "account2:\n";
    account2.output(cout);
    return 0;
}

// Определения перегруженной функции-члена set.

void BankAccount::set(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }

    balance = dollars + 0.01*cents;
    interest_rate = rate;
}

void BankAccount::set(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
```

продолжение

Листинг 6.4 (*продолжение*)

```
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }

    balance = dollars;
    interest_rate = rate;
}

// В определении одной функции-члена вызывается другая функция-член.
void BankAccount::update()
{
    balance = balance + fraction(interest_rate)*balance;
}

double BankAccount::fraction(double percent_value)
{
    return (percent_value/100.0);
}

double BankAccount::get_balance()
{
    return balance;
}

double BankAccount::get_rate()
{
    return interest_rate;
}

// Используем библиотеку классов iostream.
// В потоковом параметре можно передать как cout, так и файловый выходной поток.
void BankAccount::output(ostream& outs)
 {
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
    outs << "Account balance $" << balance << endl;
    outs << "Interest rate " << interest_rate << "%" << endl;
}
```

Пример диалога

```
Start of Test:
account1 initial statement:
Account balance $123.99
Interest rate 3.00%
account1 with new setup:
Account balance $100.00
Interest rate 5.00%
account1 after update:
Account balance $105.00
Interest rate 5.00%
account2:
Account balance $105.00
Interest rate 5.00%
```

Прежде всего, обратите внимание на то, что у класса BankAccount имеется закрытая функция-член fraction. Поскольку это закрытая функция, ее нельзя вызывать из функции main и любых других функций программы, не являющихся членами класса BankAccount. Она может быть вызвана только другими функциями-членами этого класса. Функция-член fraction выполняет лишь вспомогательную роль и к использованию класса извне никакого отношения не имеет: она позволяет упростить и сделать более понятным определение функции-члена update. Функция-член fraction принимает в качестве аргумента одно значение, представляющее собой процентную ставку (заданную в процентах), например 10.0 для 10 %, и преобразует его в дробное значение (здесь — 0.10). Подобное представление процентной ставки наиболее удобно для выполнения вычислений. В частности, если на счету находится $100,00 и процентная ставка равна 10 %, для этой суммы следует начислить $100,00 * 0.10, то есть $10,00.

Вызывая в функции main открытые функции-члены класса, скажем функцию update, нужно обязательно указывать имя объекта, отделив его точкой, как в следующей строке:

```
account1.update();
```

Если же функция-член класса (обычно закрытая) вызывается из другой функции-члена класса, задается только ее имя без вызывающего объекта и оператора точка. Так, следующее определение функции-члена BankAccount::update:

```
void BankAccount::update()
{
    balance = balance + fraction(interest_rate)*balance;
}
```

включает вызов BankAccount::fraction (см. листинг 6.4).

Вызывающий объект функции-члена fraction или переменных-членов balance и interest_rate определяется только при вызове функции update. Например, вызов

```
account1.update();
```

интерпретируется следующим образом:

```
account1.balance = account1.balance + account1.fraction(account1.interest_rate)*
                   account1.balance;
```

Обратите внимание, что вызов функции-члена fraction обрабатывается в этом отношении точно так же, как ссылки на переменные-члены.

Как и классы, рассмотренные нами ранее, BankAccount включает функцию-член для вывода хранящейся в объекте информации. Указанная функция называется output. В данной программе информация выводится на экран. Однако определение класса написано так, что его можно копировать в другие программы и использовать без изменений. В любой из этих программ может потребоваться вывести информацию объекта не на экран, а в файл. Поэтому мы определили для функции-члена output формальный параметр типа ostream, чтобы она могла принимать в качестве аргумента выходной поток, причем как cout, так и файловый.

Поскольку в программе из листинга 6.4 информация выводится на экран, вызовы функции output выполняются таким образом:

```
account1.output(cout);
```

Если бы мы захотели вывести данные в файл, а не на экран, нужно было бы сначала создать выходной файловый поток и соединить с ним файл, как рассказывалось в главе 5. Следующая команда:

```
account1.output(fout);
```

задает запись в файл информации, касающейся объекта account1, для потока fout.

С помощью объекта BankAccount определяется банковский счет, на котором имеется некоторый баланс и по которому производятся процентные начисления. Баланс и процентная ставка для начислений задаются с помощью функции-члена set. Обратите внимание, что эта функция перегружена — класс включает две ее версии. В одной из них имеется три формальных параметра, а в другой — только два. Обе версии содержат формальный параметр типа double для процентной ставки, но баланс в них представлен по-разному. В первой версии баланс задается с помощью двух формальных параметров для представления долларов и центов. Во второй баланс указывается в одном параметре, представляющем доллары, и предполагается, что количество центов равно нулю. Вторая версия функции-члена set удобнее, поскольку большинство людей открывая счет, кладут на него некоторую «круглую» сумму, например $1000, без центов. Заметьте, что концепция перегрузки функций-членов класса не содержит для вас абсолютно ничего нового: они перегружаются подобно обычным функциям.

Резюме: некоторые свойства классов

Классы обладают теми же свойствами, что и структуры, а также рядом дополнительных свойств, связанных с поддержкой функций-членов. Ниже перечислены важные моменты, о которых следует помнить при разработке и использовании классов.

- Членами классов могут быть переменные и функции.
- Члены классов (функции и переменные) могут быть закрытыми или открытыми.
- Обычно все переменные-члены класса объявляются как закрытые.
- Закрытые члены класса можно использовать только в определениях функций-членов того же класса.
- Функции-члены класса перегружаются так же, как и обычные функции.
- Типом переменной-члена класса может быть другой класс.
- Типом формального параметра функции может быть класс (см. упражнения для самопроверки 19 и 20).
- Функция может возвращать объект. Это означает, что типом возвращаемого функцией значения может быть класс (см. упражнение для самопроверки 21).

Структуры и классы

Структуры обычно используются в тех случаях, когда определяется простой набор переменных (не содержащий функций), все члены которого открыты. Но на самом деле в C++ структура может включать закрытые переменные-члены, а также открытые и закрытые функции-члены. В соответствии с изложенным выше материалом, структура C++ выполняет те же функции и обладает теми же свойствами, что и класс. Их различия в C++ определяются свойствами классов, о которых мы еще не говорили.

Однако традиционно структуры применяются так, как описано выше в этой главе, а не как эквиваленты классов с другим синтаксисом. Концепция структуры широко используется в разных языках программирования и, как правило, структуры в этих языках не могут содержать ни функций, ни закрытых переменных. Мы советуем вам следовать общепринятой практике и применять структуры лишь в качестве наборов логически объединенных открытых переменных.

Упражнения для самопроверки

19. Приведите определение функции, соответствующее следующему объявлению (класс BankAccount определен в листинге 6.4):

```
double difference(BankAccount account1, BankAccount account2);
// Предусловие: установлены значения объектов account1 и account2
// (то есть присвоены значения их переменным-членам);
// возвращает баланс, равный разности account1 - account2.
```

20. Напишите определение функции, соответствующее следующему объявлению (класс BankAccount определен в листинге 6.4).

```
void double_update(BankAccount& the_account);
// Предусловие: установлено значение объекта the_account
// (то есть присвоены значения его переменным-членам).
// Постусловие: баланс счета изменен – к нему добавлены
// начисления за два года.
```

Подсказка. Воспользуйтесь функцией-членом.

21. Приведите определение функции, соответствующее следующему объявлению (класс BankAccount определен в листинге 6.4):

```
BankAccount new_account(BankAccount old_account);
// Предусловие: установлено значение объекта old_account
// (то есть присвоены значения его переменным-членам);
// возвращает новый объект типа BankAccount с нулевым балансом
// и такой же процентной ставкой, как в объекте old_account.
```

Например, программа, в которой определена эта функция, может содержать следующий код:

```
BankAccount account3, account4;
account3.set(999, 99, 5.5);
account4 = new_account(account3);
account4.output(cout);
```

выводящий такие данные:

```
Account balance $0.00
Interest rate 5.50%
```

Инициализация с помощью конструкторов

Очень часто все или некоторые переменные-члены объекта должны инициализироваться при его объявлении. Далее в этой книге вы узнаете, какие еще операции требуется выполнить при объявлении объекта, хотя самой распространенной из них, безусловно, является именно инициализация переменных. Для выполнения такого рода операций в C++ предусмотрено специальное средство. Определяя класс, программист может использовать особую функцию-член, называемую *конструктором*. Конструктор автоматически вызывается при объявлении объекта данного класса. В нем можно выполнять любые действия, в частности инициализировать переменные-члены только что созданного объекта. В целом конструктор определяется так же, как и любая другая функция-член класса, но существуют два исключения.

- Конструктор должен иметь то же имя, что и класс. Например, если класс назван BankAccount, его конструктор должен носить это же имя.
- Конструктор не может возвращать значение. Более того, в начале его объявления не только не указывается тип возвращаемого значения, но отсутствует даже ключевое слово void, необходимое в заголовке любой другой функции.

Допустим, нам нужно определить конструктор для инициализации баланса и процентной ставки объектов класса BankAccount, определенного в листинге 6.4. С этой целью следует переопределить наш класс таким образом (для экономии места некоторые комментарии опущены):

```
class BankAccount
{
public:
BankAccount(int dollars, int cents, double rate);
    // Инициализирует баланс счета значением $dollars.cents
    // и инициализирует процентную ставку (rate).

    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();

    double get_balance();
    double get_rate();
    void output(ostream& outs);
private:
    double balance;
    double interest_rate;
    double fraction(double percent);
};
```

Заметьте, что конструктор назван BankAccount — так же, как и класс. Кроме того, объявление функции-конструктора BankAccount не начинается ни ключевым словом void, ни именем какого-нибудь другого типа. И наконец, обратите внимание, что конструктор помещен в открытую секцию определения класса. Как правило, конструктор делают открытым членом класса. Если объявить все конструкторы закрытыми, объекты этого класса нельзя будет объявлять, что сделает класс совершенно бесполезным.

После переопределения класса BankAccount его объекты можно объявлять и инициализировать следующим образом:

```
BankAccount account1(10, 50, 2.0), account2(500, 0, 4.5);
```

Согласно комментариям к объявлению конструктора приведенная выше строка объявит объект account1, присвоит переменной account1.balance значение 10.50, а переменной account1.interest_rate — 2.0. Иными словами, объект account1 будет инициализирован так, что станет представлять банковский счет со значениями баланса $10,50 и процентной ставки 2,0 %. Аналогичным образом, объект account2 инициализируется значениями баланса $500,00 и процентной ставки 4,5 %. А происходит это так. Сначала объявляется объект account1, далее вызывается конструктор BankAccount с тремя аргументами: 10, 50 и 2.0. Затем объявляется account2 и вызывается конструктор BankAccount с аргументами 500, 0 и 4.5. Результат эквивалентен следующему псевдокоду (который не является допустимым кодом C++):

```
BankAccount account1, account2;     // Эта строка некорректна, но ее можно исправить.
account1.BankAccount(10, 50, 2.0);  // СОВЕРШЕННО НЕДОПУСТИМО
account2.BankAccount(500, 0, 4.5);  // СОВЕРШЕННО НЕДОПУСТИМО
```

Как показывают комментарии, эти три строки кода нельзя включить в программу. Первую еще можно сделать допустимой, но следующие два вызова конструктора совершенно неправильны. Конструктор запрещается вызывать как обычную функцию-член. Но несмотря на это смысл трех приведенных строк вполне понятен — они отражают то, что автоматически происходит при следующем объявлении объектов account1 и account2:

```
BankAccount account1(10, 50, 2.0), account2(500, 0, 4.5);
```

Определение конструктора подобно определению любой другой функции-члена класса. Например, если модифицировать определение класса BankAccount, введя в него объявление описанного выше конструктора, нужно будет добавить и определение этого конструктора:

```
BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    balance = dollars + 0.01*cents;
    interest_rate = rate;
}
```

Поскольку имена класса и конструктора совпадают, в заголовке функции имя BankAccount повторяется дважды: перед оператором :: (как имя класса) и после него (как имя функции-конструктора). Кроме того, заметьте, что в заголовке определения конструктора отсутствует ключевое слово void или тип возвращаемого значения. В остальном конструктор определяется подобно любой другой функции-члену класса.

Конструктор можно перегружать, как любую функцию-член (например, функцию BankAccount::set из листинга 6.4). На практике конструкторы часто перегружают, чтобы объекты можно было инициализировать разными способами. Так, в программе, представленной в листинге 6.5, класс BankAccount переопределен еще раз: в него включены три версии конструктора. Перегруженный конструктор BankAccount может иметь три аргумента, два или ни одного.

Допустим, что в объявлении объекта задано только два аргумента конструктора:

```
BankAccount account1(100, 2.3);
```

В этом случае объект account1 представляет счет со значениями баланса $100,00 и процентной ставки 2,3 %.

Если же не задано ни одного аргумента, например:

```
BankAccount account2;
```

инициализированный объект account2 представляет счет со значениями баланса $0,00 и процентной ставки 0,0 %. Обратите внимание, что при отсутствии аргументов у конструктора круглые скобки в объявлении объекта не нужны. Следующее объявление:

```
BankAccount account2();   // НЕПРАВИЛЬНО
```

является неверным.

Из обновленного определения класса BankAccount в листинге 6.5 удалена функция-член set. При таком разнообразии конструкторов никакая другая функция для установки значений переменных-членов класса не нужна. Перегруженный конструктор BankAccount (см. листинг 6.5) выполняет все те же задачи, что и перегруженная функция-член set (входившая в состав старой версии класса в листинге 6.4).

Конструктор

Конструктор — это функция-член класса, имеющая то же имя, что и класс, и вызываемая автоматически при объявлении каждого объекта этого класса. Обычно конструкторы используются для инициализации объектов.

Листинг 6.5. Класс с конструкторами

```
// Программа, демонстрирующая использование класса BankAccount.
#include <iostream>
using namespace std;

// Это определение класса BankAccount является усовершенствованной
// версией одноименного класса, определенного в листинге 6.4.
// Класс для банковского счета:
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    // Инициализирует баланс счета значением $dollars.cents
    // и процентную ставку счета значением аргумента rate.
```

```
    BankAccount(int dollars, double rate);
    // Инициализирует баланс счета значением $dollars.00
    // и процентную ставку счета значением аргумента rate.

    // Конструктор, используемый по умолчанию.
    BankAccount();
    // Инициализирует баланс счета значением $0.00
    // и процентную ставку счета значением 0.0 %.

    void update();
    // Постусловие: к балансу счета добавлены процентные
    // начисления за один год.

    double get_balance();
    // Возвращает значение текущего баланса счета.

    double get_rate();
    // Возвращает текущую процентную ставку счета.

    void output(ostream& outs);
    // Предусловие: если outs – выходной файловый поток,
    // он уже соединен с файлом.
    // Постусловие: значения баланса счета и процентной ставки
    // записаны в поток outs.
private:
    double balance;
    double interest_rate;

    double fraction(double percent);
    // Преобразует проценты в дробь. Функция fraction(50.3) возвращает значение 0.503.
};

int main()
{
// При объявлении объекта account2 вызывается конструктор, используемый
// по умолчанию. Обратите внимание на отсутствие круглых скобок.
    BankAccount account1(100, 2.3), account2;

    cout << "account1 initialized as follows:\n";
    account1.output(cout);
    cout << "account2 initialized as follows:\n";
    account2.output(cout);

    account1 = BankAccount(999, 99, 5.5); // Явный вызов конструктора BankAccount::BankAccount.
    cout << "account1 reset to the following:\n";
    account1.output(cout);
    return 0;
}

BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
```

продолжение

Листинг 6.5 (*продолжение*)

```
    balance = dollars + 0.01*cents;
    interest_rate = rate;
}

BankAccount::BankAccount(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    balance = dollars;
    interest_rate = rate;
}

BankAccount::BankAccount(): balance(0), interest_rate(0.0)
{
   // Тело намеренно оставлено пустым.
}
void BankAccount::output(ostream& outs)
 {
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
    outs << "Account balance $" << balance << endl;
    outs << "Interest rate" << interest_rate << "%" << endl;
}
```

Пример диалога

```
account1 initialized as follows:
Account balance $100.00
Interest rate 2.30%
account2 initialized as follows:
Account balance $0.00
Interest rate 0.00%
account1 reset to the following:
Account balance $999.99
Interest rate 5.50%
```

Конструктор без параметров в листинге 6.5 требует отдельного рассмотрения, поскольку содержит элементы, которые вам еще не знакомы. Вот его определение:

```
BankAccount::BankAccount(): balance(0), interest_rate(0.0)
{
   // Тело намеренно оставлено пустым.
}
```

Новым элементом здесь является часть первой строки после двоеточия. Эта часть определения конструктора называется *разделом инициализации*. Как показывает данный пример, раздел инициализации расположен после круглых скобок, куда помещен список параметров, и перед открывающей фигурной скобкой, за которой следует тело функции. Он состоит из двоеточия и следующего за ним списка части или всех переменных-членов класса, разделенных запятыми. После имени

каждой переменной в скобках указано значение, которым она инициализируется. Приведенное выше определение конструктора эквивалентно следующему:

```
BankAccount::BankAccount()
{
    balance = 0;
    interest_rate = 0.0;
}
```

Тело функции в определении конструктора с инициализационным разделом не обязательно должно быть пустым. Например, приведенное далее определение конструктора с двумя параметрами:

```
BankAccount::BankAccount(int dollars, double rate)
                    : balance(dollars), interest_rate(rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
}
```

эквивалентно определению из листинга 6.5.

Обратите внимание, что инициализационными значениями в данном случае являются значения параметров.

Конструктор автоматически вызывается при объявлении объекта класса, но может быть вызван и после объявления объекта для установки значений его переменных-членов. Еще один способ вызова конструктора состоит в следующем. Вызов конструктора создает анонимный объект с новыми значениями. Анонимный объект — это объект, не присвоенный (пока) ни одной переменной. Его можно присвоить именованному объекту, то есть переменной класса. Так, следующая строка содержит вызов конструктора класса BankAccount, создающий анонимный объект со значениями баланса $999,99 и процентной ставки 5,5 %:

```
account1 = BankAccount(999, 99, 5.5);
```

Как нетрудно теперь догадаться, конструктор действует в качестве функции, возвращающей объект того класса, в который она входит[1].

Совет программисту: всегда определяйте конструктор, используемый по умолчанию

C++ не всегда генерирует для созданного программистом класса конструктор, используемый по умолчанию. Если не определить для класса ни одного конструктора, компилятор сгенерирует применяемый по умолчанию конструктор, который ничего не делает. Этот конструктор вызывается при объявлении объектов

[1] В определение класса можно включить не только конструктор, но и функцию set. Вызов этой функции является более эффективным способом изменения значений переменных-членов объекта, поскольку в случае вызова конструктора всегда создается новый объект, а при вызове названной функции только изменяются значения переменных-членов существующего объекта.

класса. Если же определить хотя бы один конструктор класса, компилятор не станет генерировать никаких других конструкторов. Встретив очередное объявление объекта этого класса, C++ начнет искать подходящее определение конструктора. Когда в объявлении объекта не окажется аргументов для конструктора, C++ станет искать конструктор, используемый по умолчанию, и если программист его не определил, компилятор ничего не найдет.

Вызов конструктора

Конструктор вызывается автоматически при объявлении объекта, но в этом объявлении должны быть заданы все необходимые для конструктора аргументы. Кроме того, конструктор можно вызвать явно для создания нового объекта, присваиваемого переменной класса.

Синтаксис объявления объекта при наличии конструктора имеет следующий вид:

имя_класса имя_обьекта(аргументы_конструктора);

Вот пример:

```
BankAccount account1(100, 2.3);
```

А синтаксис явного вызова конструктора выглядит так:

объект = имя_конструктора(аргументы_конструктора);

Например:

```
account1 = BankAccount(200, 3.5);
```

Предположим, что мы определили класс следующим образом:

```
class SampleClass
{
public:
// Конструктор с двумя параметрами
    SampleClass(int parameter1, double parameter2);
    void do_stuff();
private:
    int data1;
    double data2;
};
```

Правильный способ объявления объекта класса `SampleClass` таков:

```
SampleClass my_object(7, 7.77);
```

Однако объявление

```
SampleClass your_object;
```

в данном случае недопустимо.

Компилятор интерпретирует его как вызов конструктора без аргументов, а поскольку такого конструктора нет, он выдает сообщение об ошибке. Необходимо либо задать в объявлении объекта `your_object` два аргумента, либо добавить еще один конструктор, не имеющий аргументов.

Конструктор, который можно вызвать без аргументов, называется *конструктором, используемым по умолчанию*: он вызывается в самом простом случае — когда

аргументы не заданы. И поскольку велика вероятность того, что иногда объекты класса будут объявляться без аргументов, лучше всегда определять конструктор, используемый по умолчанию. Следующая переопределенная версия класса SampleClass содержит такой конструктор:

```
class SampleClass
{
public:
    SampleClass(int parameter1, double parameter2);
    SampleClass();   // Конструктор, используемый по умолчанию.
    void do_stuff();
private:
    int data1;
    double data2;
};
```

При переопределении класса SampleClass указанным образом приведенное выше объявление объекта your_object станет допустимым.

Если вы не хотите, чтобы используемый по умолчанию конструктор инициализировал переменные-члены класса, можно оставить его тело пустым. Следующий конструктор:

```
SampleClass::SampleClass()
{
    // Ничего не делать.
}
```

вполне допустим, но его единственная задача — «осчастливить» компилятор.

Ловушка: конструкторы без аргументов

Если конструктор класса BankAccount имеет два формальных параметра, с его помощью объект этого класса объявляется следующим образом:

```
BankAccount account1(100, 2.3);
```

По аналогии с обычными функциями можно подумать, что объявление объекта с помощью конструктора без аргументов выполняется так:

```
BankAccount account2();   // ЭТО ВЫЗОВЕТ ПРОБЛЕМЫ
```

Но для конструктора это неверно. Кроме того, данный оператор не вызовет сообщения об ошибке после компиляции, поскольку синтаксически он вполне правилен, хотя и делает не то, что мы хотим. Компилятор решит, что эта строка является объявлением функции account2, не имеющей аргументов и возвращающей значение типа BankAccount.

Итак, объявление объекта без аргументов для конструктора не должно содержать круглых скобок. Правильное объявление объекта account2 таково:

```
BankAccount account2;
```

Если конструктор явно вызывается в операторе присваивания, скобки нужны, как и при вызове обычной функции. Для класса BankAccount (см. листинг 6.5) оператор

```
account1 = BankAccount();
```

создаст новый объект, установит для него нулевой баланс и нулевую процентную ставку, а также присвоит этот объект переменной account1.

Конструкторы без аргументов

В объявлении объекта без аргументов для конструктора не должны использоваться круглые скобки. Например, чтобы объявить объект и передать его конструктору два аргумента, можно написать такую строку:

```
BankAccount account1(100, 2.3);
```

Но если вы хотите воспользоваться конструктором, не имеющим аргументов, объект следует объявить так:

```
BankAccount account1;
```

С помощью следующего оператора:

```
BankAccount account1();   // НЕВЕРНОЕ ОБЪЯВЛЕНИЕ
```

нельзя объявить объект.

Этот оператор объявляет функцию account1 без аргументов, возвращающую объект типа BankAccount.

Упражнения для самопроверки

22. Предположим, что программа содержит следующее определение класса (и определения функций-членов):

```
class YourClass
{
public:
    YourClass(int new_info, char more_new_info);
    YourClass();
    void do_stuff();
private:
    int information;
    char more_information;
};
```

Какие из приведенных ниже операторов являются допустимыми?

```
YourClass an_object(42, 'A');
YourClass another_object;
YourClass yet_another_object();
an_object = YourClass(99, 'B');
an_object = YourClass();
an_object = YourClass;
```

23. Как бы вы изменили определение класса DayOfYear из листинга 6.3, чтобы этот класс содержал две версии (перегруженного) конструктора? Первая версия имеет два формальных параметра типа int (один для месяца и один для дня) и устанавливает закрытые переменные-члены так, чтобы они хранили месяц и день. Вторая не имеет формальных параметров и устанавливает дату, представляющую 1 января. Ни в одном конструкторе не используйте раздел инициализации.

24. Выполните предыдущее упражнение, но на этот раз включите в каждый конструктор раздел инициализации и инициализируйте в нем все переменные-члены.

6.3. Абстрактные типы данных

Все мы знаем — «Таймс» знает — но делаем вид, будто не знаем.

Вирджиния Вульф

Каждый простой тип данных представляет определенные значения, например тип `int` представляет такие значения, как `0`, `1`, `-1`, `2` и т. д. Поэтому можно подумать, что тип данных состоит только из таких значений, однако это совершенно неверно. Помимо самих значений, важны и выполняемые над ними операции. К числу операций, производимых над данными типа `int`, относятся такие: сложение (+), вычитание (-), умножение (*), деление (/) и еще несколько операций и предопределенных библиотечных функций. Таким образом, *тип данных* — это не просто набор значений, а еще и набор определенных для них базовых операций.

Если программист, использующий некоторый тип данных, не имеет доступа к коду, которым данный тип задается, то это *абстрактный* тип данных. В частности, абстрактными являются такие предопределенные типы, как `int` и `double`, поскольку вы не знаете, как для них реализованы операции наподобие сложения или умножения. А если бы и знали, все равно не использовали бы эту информацию ни в одной программе на C++.

Типы, определяемые программистами, скажем структуры и классы, в общем случае не являются абстрактными. Если определяемый программистом тип разработан плохо, то с ним неудобно работать, а программу с его применением трудно понять и модифицировать. Избежать этого можно, определяя все типы как абстрактные, что предполагает использование классов. Чтобы класс был абстрактным типом, его определение должно быть выполнено в соответствии с правилами, которые описаны в следующем разделе.

Классы как абстрактные типы данных

В отличие от простых предопределенных типов данных, таких как `int` или `char`, класс представляет собой тип, определяемым программистом. Значением типа класса является набор значений его переменных-членов. Например, значение типа `BankAccount`, определенного в листинге 6.5, состоит из двух чисел типа `double`. Для удобства мы повторим здесь это определение (опустив комментарии):

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    BankAccount(int dollars, double rate);
    BankAccount();
    void update();
```

```
    double get_balance();
    double get_rate();
    void output(ostream& outs);
private:
    double balance;
    double interest_rate;
    double fraction(double percent);
};
```

Программисту, применяющему тип BankAccount, не нужно знать, как реализовано определение функции BankAccount::update или любой другой функции-члена класса BankAccount. Мы будем использовать следующее определение функции-члена BankAccount::update:

```
void BankAccount::update()
{
    balance = balance + fraction(interest_rate)*balance;
}
```

Однако можно отказаться от закрытой функции-члена fraction и задействовать в определении функции update чуть более сложную формулу.

```
void BankAccount::update()
{
    balance = balance + (interest_rate/100.0)*balance;
}
```

Программисту, который использует класс, совершенно безразлично, каким из двух приведенных выше способов реализована функция update, поскольку действие обеих функций абсолютно одинаково. Ему также не важно, как реализованы внутри класса значения даты. Мы решили реализовать их как значения типа double. Если vacation_savings — объект типа BankAccount, его значение состоит из двух значений типа double, хранящихся в следующих двух переменных-членах:

```
vacation_savings.balance
vacation_savings.interest_rate
```

Однако значение объекта vacation_savings следует представлять себе не в виде пары чисел типа double, таких как 1.3546e+2 и 4.5, а в виде единой записи:

```
Баланс на счету $135.46
Процентная ставка 4.5 %
```

Именно поэтому наша реализация функции BankAccount::output выводит значение класса в таком формате.

То, что мы решили реализовать данное значение типа BankAccount в виде двух значений типа double, 1.3546e+2 и 4.5, — подробность реализации класса. Мы могли бы реализовать его как два значения типа int, 135 и 46 (для долларовой и центовой составляющих баланса), и одно значение типа double — 0.045 (представленное в виде дроби значение процентной ставки 4,5 %). Если реализовать класс BankAccount данным способом, его открытые члены не изменятся, а закрытые будут модифицированы следующим образом:

```
class BankAccount
{
public:
```

```
    ... Эта часть кода остается прежней. ...
private:
    int dollars_part;
    int cents_part;
    double interest_rate;
    double fraction(double percent);
};
```

Соответственно, придется изменить определения функций-членов, что совсем не сложно. Например, определение функции get_balance и одной из версий конструктора можно изменить так:

```
double BankAccount::get_balance()
{
    return (dollars_part + 0.01*cents_part);
}

BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    dollars_part = dollars;
    cents_part = cents;
    interest_rate = rate;
}
```

Подобным образом должна быть переопределена каждая функция-член класса, оперирующая значениями баланса и процентной ставки.

Обратите внимание, что даже если пользователь считает баланс единственным числом, это не означает, что он и в самом деле должен быть реализован как одно число типа double. Так, баланс может быть представлен парой чисел типа int. Программисту, который пользуется типом BankAccount, нет необходимости знать, как реализованы значения этого типа.

На основании полученной в этой главе информации о типе BankAccount можно сформулировать основной принцип определения класса как абстрактного типа данных. Суть его заключается в том, что использование типа в программе не должно зависеть от деталей его реализации. Иными словами, изменение реализации класса не должно требовать внесения изменений в использующие его программы. Вот три правила, которые помогут это обеспечить.

- Сделайте закрытыми все переменные-члены класса.
- Реализуйте каждую из необходимых программисту базовых операций с объектами класса в виде открытой функции-члена и подробно опишите, как этой функцией пользоваться.
- Сделайте закрытыми все вспомогательные функции-члены.

В главах 8 и 9 описывается альтернативный подход к определению абстрактных типов данных, но приведенные правила указывают единственный универсальный способ обеспечения абстрактности класса.

Интерфейс абстрактного типа служит для использования данного типа в программе. Если абстрактный тип данных является классом C++, его интерфейс составляют открытые функции-члены этого типа и комментарии, поясняющие, как ими пользоваться.

Реализация абстрактного типа — это скрытый от глаз программиста код данного типа. Такой код состоит из закрытых членов класса и определения открытых функций-членов класса. Хотя для выполнения программы, в которой используется абстрактный тип, его реализация необходима, для написания функции `main` и других функций программы, не являющихся членами этого типа, она не требуется. Вы, наверное, заметили явную аналогию с принципом «черного ящика» (см. главы 3 и 4). Подобно реализации обычной функции, реализация абстрактного типа данных должна быть «черным ящиком», заглядывать в который не нужно никому, кроме разработчика этого типа.

В главе 9 мы расскажем о том, как поместить интерфейс и реализацию абстрактного типа в разные файлы и отделить их от программ, где этот тип применяется. Тогда программист, который пользуется таким абстрактным типом, и в самом деле не увидит его реализацию. А пока мы будем помещать реализацию абстрактных типов в тот же файл, что и программу, считая при этом интерфейс (то есть `public`-секцию определения класса) и реализацию (то есть `private`-секцию определения класса и определения функций-членов класса) отдельными частями абстрактного типа. Мы будем стараться создавать абстрактные типы таким образом, чтобы программисту достаточно было их интерфейса и не требовалось вникать в детали реализации. А для этого следует только обеспечить возможность изменения реализации абстрактного типа без необходимости вносить какие-либо изменения в другие части программы. Данную концепцию иллюстрирует пример, описанный в следующем разделе.

Таким образом, самым очевидным преимуществом классов, представляющих собой абстрактные типы, является возможность изменения реализации абстрактного типа без необходимости вносить изменения в другие части программы. Но это не единственное их достоинство. В частности, работу над программой, классы которой определяются как абстрактные типы, можно разделить между несколькими программистами так, чтобы каждый класс разрабатывался одним программистом, а использовался другими, причем с минимальными затратами времени на освоение. Это значительно облегчает разработку и отладку программы.

Пример: альтернативная реализация класса BankAccount

В листинге 6.6 приведена альтернативная реализация класса `BankAccount`, описанного в предыдущем разделе. В этой версии данные банковского счета представлены тремя значениями: долларовая часть баланса, центовая часть баланса и процентная ставка.

Обратите внимание, что хотя обе реализации, приведенные в листингах 6.5 и 6.6, включают переменную-член `interest_rate`, ее значение в них хранится по-разному.

Если процентная ставка составляет, например, 4,7 %, в реализации из листинга 6.5 (которая подобна реализации, приведенной в листинге 6.4) значение переменной interest_rate равно 4.7. В альтернативной реализации (см. листинг 6.6) значение переменной interest_rate равно 0.047, то есть процентная ставка хранится в виде дроби. Главным отличием новой реализации класса от предыдущей является то, что как только программа задает значение процентной ставки, оно немедленно преобразуется в дробь с помощью функции-члена fraction. Эта функция используется в определениях конструкторов, но зато она больше не нужна в функции-члене update, поскольку значение переменной-члена interest_rate уже преобразовано в дробь. В прежней реализации (листинг 6.5) ситуация была прямо противоположной: функция fraction использовалась в функции update и не применялась в конструкторах.

Листинг 6.6. Альтернативная реализация класса BankAccount

```
// Демонстрирует альтернативную реализацию класса BankAccount.
#include <iostream>
#include <cmath>
using namespace std;

// Обратите внимание, что открытые члены класса BankAccount
// выглядят и действуют точно так же, как в программе из листинга 6.5.

// Класс для банковского счета:
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    // Инициализирует баланс счета значением $dollars.cents
    // и процентную ставку счета значением аргумента rate.

    BankAccount(int dollars, double rate);
    // Инициализирует баланс счета значением $dollars.00
    // и процентную ставку счета значением аргумента rate.

    BankAccount();
    // Инициализирует баланс счета значением $0.00
    // и процентную ставку счета значением 0.0 %.

    void update();
    // Постусловие: к балансу счета прибавлены процентные
    // начисления за один год.

    double get_balance();
    // Возвращает значение текущего баланса счета.

    double get_rate();
    // Возвращает значение текущей процентной ставки счета (в процентах).


    void output(ostream& outs);
    // Предусловие: если outs – выходной файловый поток,
    // он уже соединен с файлом.
```

продолжение

Листинг 6.6 (*продолжение*)

```
    // Постусловие: значения баланса счета и процентной ставки
    // записаны в поток outs.
private:
    int dollars_part;
    int cents_part;
    double interest_rate;   // В виде дроби, например 0.057 для 5,7 %.

    double fraction(double percent);
    // Преобразует проценты в дробь. Например, функция fraction(50.3)
    // возвращает значение 0.503.

    double percent(double fraction_value);   // Новая функция.
    // Преобразует дробь в проценты. Например, функция percent(0.503)
    // возвращает значение 50.3.
};

int main()
// Поскольку тело функции main идентично телу, приведенному
// в листинге 6.5, выводимые на экран данные будут такими же.
{
    BankAccount account1(100, 2.3), account2;

    cout << "account1 initialized as follows:\n";
    account1.output(cout);
    cout << "account2 initialized as follows:\n";
    account2.output(cout);

    account1 = BankAccount(999, 99, 5.5);
    cout << "account1 reset to the following:\n";
    account1.output(cout);
    return 0;
}
BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    dollars_part = dollars;
    cents_part = cents;
    interest_rate = fraction(rate);
}
// В предыдущей реализации этого абстрактного типа закрытая функция-член
// fraction применялась в определении функции update. В данной
// реализации она используется в определении конструкторов.
BankAccount::BankAccount(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
```

```
    dollars_part = dollars;
    cents_part = 0;
    interest_rate = fraction(rate);
}

BankAccount::BankAccount() : dollars_part(0), cents_part(0), interest_rate(0.0)
{
    // Тело намеренно оставлено пустым.
}

double BankAccount::fraction(double percent_value)
{
    return (percent_value/100.0);
}

// Используем библиотеку классов cmath:
void BankAccount::update()
{
    double balance = get_balance();
    balance = balance + interest_rate*balance;
    dollars_part = floor(balance);
    cents_part = floor((balance - dollars_part)*100);
}

double BankAccount::get_balance()
{
    return (dollars_part + 0.01*cents_part);
}

double BankAccount::percent(double fraction_value)
{
    return (fraction_value*100);
}

double BankAccount::get_rate()
{
    return percent(interest_rate);
}

// Используем библиотеку классов iostream:
void BankAccount::output(ostream& outs)
{
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
// Новые определения функций-членов get_balance и get_rate
// гарантируют, что данные всегда будут выводиться корректно.
    outs << "Account balance $" << get_balance() << endl;
    outs << "Interest rate " << get_rate() << "%" << endl;
}
```

Хотя мы модифицировали определения закрытых членов класса BankAccount, в открытом разделе определения класса ничего не изменилось. Открытые функции-члены объявлены так же и ведут себя, как и в старой версии класса, приведенной

в листинге 6.5. Например, хотя в этой новой реализации проценты хранятся в виде дроби (в частности, 4,7 % хранится в виде значения `0.047`), функция-член `get_rate` возвращает значение `4.7`, совершенно так же, как и в старой реализации. Подобным образом функция-член `get_balance`, как и прежде, возвращает одно значение типа `double`, представляющее баланс в виде числа с десятичной точкой. И тот факт, что теперь баланс хранится в двух целочисленных переменных, а не в одной переменной типа `double`, не мешает функции `get_balance` представить его аналогично тому, как он был представлен в старой версии класса.

Обратите внимание, что при модификации класса мы по-разному поступаем с его открытыми и закрытыми членами. Если нужно сохранить интерфейс абстрактного класса, чтобы не пришлось вносить изменения в использующие его программы, то открытые члены класса должны остаться неизменными. Однако можно удалять, добавлять и изменять любые закрытые функции-члены класса. В этом примере мы добавили одну дополнительную закрытую функцию `percent`, действие которой обратно действию функции `fraction`. Если функция `fraction` преобразует проценты в дробь, то функция `percent` преобразует дробь обратно в проценты. Например, функция `fraction(4.7)` возвращает значение `0.047`, а функция `percent(0.047)` — значение `4.7`.

Сокрытие информации

Мы рассматривали сокрытие информации, когда описывали функции в главе 3. Там говорилось, что в отношении функций сокрытие информации означает следующее: пользовательские функции необходимо создавать так, чтобы ими можно было пользоваться как «черным ящиком», то есть чтобы применяющему их программисту не нужно было знать о том, как они реализованы. Ему достаточно объявления функции и сопутствующего комментария, поясняющего, как использовать данную функцию. Принцип сокрытия информации также применим к классам и выражается в определении классов как абстрактных типов, то есть в использовании закрытых переменных и закрытых функций.

Упражнения для самопроверки

25. Если класс C++ определяется как абстрактный тип, какими следует объявить его переменные-члены и функции-члены: открытыми или закрытыми?
26. Если класс C++ определяется как абстрактный тип, какие его элементы считаются частью интерфейса этого типа, а какие — частью его реализации?
27. Предположим, ваш коллега определил абстрактный тип как класс C++, следуя правилам, описанным в разделе 6.3. Вам поручено создать программу на основе этого типа. Иначе говоря, нужно написать функцию `main` и другие использующиеся в ней функции, не являющиеся членами класса. Полное определение абстрактного типа является очень длинным, а времени у вас немного. Какие его части следует прочитать, а какие можно проигнорировать?
28. Перепишите конструкторы с двумя или тремя параметрами из программы, представленной в листинге 6.6, чтобы все переменные-члены устанавливались в разделе инициализации.

Резюме

- Структура может использоваться для объединения данных разных типов в одно составное значение.
- Класс служит для объединения данных и функций в один составной объект.
- Переменная-член или функция-член класса могут быть как открытыми, так и закрытыми. Открытые члены класса могут применяться вне этого класса, а закрытые — только в определениях функций-членов данного класса.
- Формальные параметры функции могут иметь тип класса или тип структуры. Функция может возвращать значение типа класса или структуры.
- Функция-член класса может быть перегружена так же, как обычная функция.
- Конструктор — это функция-член класса, вызываемая автоматически при объявлении объекта класса. Конструктор должен иметь то же имя, что и его класс.
- Тип данных состоит из набора значений и набора базовых операций над этими значениями.
- Тип данных называется абстрактным в том случае, если использующему его программисту не нужно вникать в детали реализации значений этого типа и операций над ними.
- Один из способов реализации абстрактного типа данных в C++ — определение класса, все переменные-члены которого являются закрытыми, а операции представляют собой открытые функции-члены.

Ответы к упражнениям для самопроверки

1. Перечисленные элементы имеют следующие типы:
   а) `double`;
   б) `double`;
   в) недопустимое выражение, тег структуры не может использоваться вместо переменной структурного типа;
   г) недопустимое выражение, переменная `savings_account` не объявлена;
   д) `char`;
   е) `TermAccount`.
2.
```
A $9.99
A $1.11
```
3. Многие компиляторы выдают непонятные сообщения об ошибках, но, как ни странно, сообщение компилятора g++ довольно информативно. Оно выглядит следующим образом:

```
g++ -fsyntax-only c6testg1.cc
probl.cc:8: semicolon missing after declaration of
'Stuff'
```

```
prob1.cc:8: extraneous 'int' ignored
prob1.cc:8: semicolon missing after declaration of
'struct Stuff'
```

(«После объявления 'Stuff' пропущена точка с запятой».)

4. Объявление переменной x имеет вид:

```
A x = {1,2};
```

5. а) слишком мало значений — это не ошибка. После инициализации значения членов структуры станут такими: due_date.day = 21, due_date.month = 12, due_date.year = 0. Члены структуры, для которых значения не заданы, будут инициализированы нулевыми значениями соответствующих типов;

   б) после инициализации значения членов структуры будут такими: due_date.month = 12, due_date.day = 21, due_date.year = 2002;

   в) ошибка: слишком много значений;

   г) это может быть логической ошибкой. Программист задал для года всего две цифры, тогда как их должно быть четыре. В результате, в зависимости от того, что программа делает с этим значением, в ней, скорее всего, будет использоваться неверная дата.

6.
```
struct EmployeeRecord
{
    double wage_rate;
    int vacation;
    char status;
};
```

7.
```
void read_shoe_record(ShoeType& new_shoe)
{
    cout << "Enter shoe style (one letter): ";
    cin >> new_shoe.style;
    cout << "Enter shoe price $";
    cin >> new_shoe.price;
}
```

8.
```
ShoeType discount(ShoeType old_record)
{
    ShoeType temp;
    temp.style = old_record.style;
    temp.price = 0.90*old_record.price;
    return temp;
}
```

9.
```
struct StockRecord
{
    ShoeType shoe_info;
    Date arrival_date;
};
```

10.
```
StockRecord aRecord;
aRecord.arrival_date.year = 2004;
```

11.
```
void DayOfYear::input()
{
    cout << "Enter month as a number: ";
```

```
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
}
```

12.
```
void Temperature::set(double new_degrees, char new_scale)
{
    degrees = new_degrees;
    scale = new_scale;
}
```

13. И оператор точка (.), и оператор :: используются с именами членов для указания класса или структуры, к которым эти члены принадлежат. Если класс DayOfYear определен так, как показано в листинге 6.2, и today — объект данного класса, то для доступа к переменной-члену month может применяться оператор точка: today.month. В определении функции-члена оператор :: позволяет указать компилятору, что эта функция объявлена в классе, имя которого указано перед оператором ::. Различие данных операторов отражено и в их полных называниях: оператор точка именуется оператором прямого доступа к члену класса, а оператор :: — оператором доступа к члену класса по имени класса.

14. Если вам непонятны логические выражения в некоторых операторах if, перед выполнением данного упражнения ознакомьтесь с главой 7.

```
void DayOfYear::check_date()
{
    if ((month < 1) || (month > 12)
        || (day < 1) || (day > 31))
    {
        cout << " Illegal date. Aborting program.\n";
        exit(1);
    }
    if (((month == 4) || (month == 6) || (month == 9)
                                      || (month == 11))
        && (day == 31))
    {
        cout << "Illegal date. Aborting program.\n";
        exit(1);
    }
    if ((month == 2) && (day > 29))
    {
        cout << "Illegal date. Aborting program.\n";
        exit(1);
    }
}
```

15.
```
hyundai.price = 4999.99;          // НЕДОПУСТИМО (Переменная-член price является закрытой.)
jaguar.set_price(30000.97);       // ДОПУСТИМО
double a_price, a_profit;         // ДОПУСТИМО
a_price = jaguar.get_price();     // ДОПУСТИМО
a_profit = jaguar.get_profit();   // НЕДОПУСТИМО (Функция get_profit является закрытой.)
a_profit = hyundai.get_profit();  // НЕДОПУСТИМО (Функция get_profit является закрытой.)
if (hyundai == jaguar)            // НЕДОПУСТИМО (Оператор == не используется с объектами.)
hyundai = jaguar;                 // ДОПУСТИМО
```

16. После внесения изменений допустимыми станут все операторы, кроме приведенного ниже:

```
if (hyundai == jaguar)   // НЕДОПУСТИМО (Нельзя использовать оператор == с объектами.)
```

17. Ключевое слово private объявляет следующие за ним члены класса как закрытые. Закрытые функции-члены доступны только из функций-членов того же класса. Значения закрытых переменных-членов могут считываться и устанавливаться лишь функциями-членами того же класса, что позволяет контролировать изменения закрытых данных объектов и предотвращать их случайное изменение.

18. а) только одна. Компилятор выдаст предупреждение в случае, если в определении класса (или структуры) не окажется ни одного открытого члена;

    б) ни одной, хотя обычно класс содержит как минимум одну секцию private:;

    в) в определении класса эта секция по умолчанию считается закрытой, то есть private;

    г) в определении структуры она по умолчанию считается открытой, то есть public.

19. Возможный правильный ответ:

```
double difference(BankAccount account1, BankAccount account2)
{
    return (account1.get_balance() - account2.get_balance());
}
```

Обратите внимание, что следующий ответ некорректен, поскольку переменная balance является закрытым членом класса:

```
double difference(BankAccount account1, BankAccount account2)
{
    return (account1.balance - account2.balance);   // НЕДОПУСТИМО
}
```

20.
```
void double_update(BankAccount& the_account)
{
    the_account.update();
    the_account.update();
}
```

Учтите, что поскольку это не функция-член, вызов функции-члена update должен содержать имя объекта и оператор точка.

21.
```
BankAccount new_account(BankAccount old_account)
{
    BankAccount temp;
    temp.set(0, old_account.get_rate());
    return temp;
}
```

22.
```
YourClass an_object(42, 'A');   // ДОПУСТИМО
YourClass another_object;       // ДОПУСТИМО
YourClass yet_another_object(); // ПРОБЛЕМА
an_object = YourClass(99, 'B'); // ДОПУСТИМО
```

```
an_object = YourClass();        // ДОПУСТИМО
an_object = YourClass;          // НЕДОПУСТИМО
```

Оператор с комментарием // ПРОБЛЕМА, строго говоря, не является недопустимым, но означает не то, что кажется на первый взгляд. Если вы думаете, что это объявление объекта yet_another_object, то ошибаетесь: перед вами объявление функции yet_another_object, которая не принимает ни одного аргумента и возвращает значение типа YourClass. Поэтому когда нужно объявить объект yet_another_object, чтобы инициализировать его используемым по умолчанию конструктором, данный оператор недопустим, а правильный оператор задается так:

```
YourClass another_object;
```

23. Вот модифицированное определение класса:

```
class DayOfYear
{
public:
    DayOfYear(int the_month, int the_day);
    // Предусловие: аргументы the_month и the_day составляют
    // допустимую дату; устанавливает дату в соответствии
    // со значениями аргументов.

    DayOfYear();
    // Инициализирует дату значением 1 января.

    void input();

    void output();

    int get_month();
    // Возвращает порядковый номер месяца: 1 для января, 2 для февраля и т. д.

    int get_day();
    // Возвращает день месяца.
private:
    void check_date();
    int month;
    int day;
};
```

Обратите внимание на то, что мы удалили объявление функции-члена set, поскольку наличие конструктора сделало ее ненужной. Для полного определения класса следует добавить такие определения функций-членов:

```
DayOfYear::DayOfYear(int the_month, int the_day)
{
    month = the_month;
    day = the_day;
    check_date();
}

DayOfYear::DayOfYear()
```

```
{
    month = 1;
    day = 1;
}
```

и удалить определение функции `DayOfYear::set`.

24. Определение класса будет таким же, как в предыдущем упражнении. А вот определение конструктора изменится:

```
DayOfYear::DayOfYear(int the_month, int the_day)
    : month(the_month), day(the_day)
{
    check_date();
}

DayOfYear::DayOfYear() : month(1), day(1)
{
    // Тело намеренно оставлено пустым.
}
```

25. Переменные-члены класса должны быть закрытыми. Функции-члены представляют собой часть интерфейса абстрактного типа (то есть являются его операциями) и поэтому должны быть открытыми. Кроме того, класс может содержать дополнительные вспомогательные функции, использующиеся только в определениях других функций-членов. Они должны быть закрытыми.

26. Все объявления закрытых переменных-членов являются частью реализации. (Класс, представляющий абстрактный тип данных, вообще не должен содержать открытых переменных-членов.) Все объявления открытых функций-членов класса (перечисленные в определении класса) и пояснительные комментарии к ним составляют интерфейс класса. Объявления закрытых функций-членов класса являются частью его реализации. Еще одна составная часть реализации класса — определение функций-членов (закрытых и открытых).

27. Вам следует прочитать только интерфейс класса, то есть объявления открытых функций-членов класса (приведенные в определении класса) и поясняющие комментарии к этим функциям. Ни объявления закрытых функций-членов, ни объявления закрытых переменных-членов, ни определения закрытых и открытых функций-членов класса читать не нужно.

28.
```
BankAccount::BankAccount(int dollars, int cents,
     double rate) : dollars_part(dollars),
     cents_part(cents), interest_rate(fraction(rate))
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout <<
            "Illegal values for money or interest rate.\n";
        exit(1);
    }
}

BankAccount::BankAccount(int dollars, double rate)
    : dollars_part(dollars), cents_part(0),
    interest_rate(fraction(rate))
```

```
{
    if ((dollars < 0) || (rate < 0))
    {
        cout <<
            "Illegal values for money or interest rate.\n";
        exit(1);
    }
}
```

Практические задания

1. Напишите программу выставления итоговых оценок, правила формирования которых описаны ниже:

   а) проведено два теста, максимальное количество баллов за каждый из них равно десяти;

   б) проведено два промежуточных экзамена и один завершающий, максимальное количество баллов за каждый из них равно 100;

   в) сумма баллов за завершающий экзамен составляет 50 % итоговой оценки, за промежуточный – 25 % и за два теста – еще 25 %. (Не забудьте выразить оценки за тесты в процентах.)

   Оценка 90 и более баллов называется A, 80 и более (но меньше 90) – B, 70 и более (но меньше 80) – C, 60 и более (но меньше 70) – D, менее 60 – F.

   Программа считывает оценки студента за экзамены и тесты, а затем выводит запись, состоящую из двух оценок за тесты, двух оценок за экзамены и итоговой оценки в виде соответствующей буквы.

2. Переопределите структуру `CDAccount` из программы, приведенной в листинге 6.1, превратив ее в класс, который должен содержать те же переменные-члены, но объявленные как закрытые. Включите в этот класс следующие четыре функции-члена: возвращающую значение исходного баланса, возвращающую значение баланса по истечении срока, пока выплаты по счету заморожены, возвращающую значение процентной ставки и возвращающую значение, соответствующее количеству месяцев до наступления срока выплаты. Кроме того, введите в состав класса функцию-член `input` с одним формальным параметром типа `istream` и функцию-член `output` с одним формальным параметром типа `ostream`. Включите определение этого класса в тестовую программу.

3. Переопределите класс `CDAccount` из проекта 2, сохранив его интерфейс, но изменив реализацию. Новая реализация должна быть подобна второй реализации класса `BankAccount`, приведенной в листинге 6.6. В ней баланс необходимо представить двумя значениями типа `int`: количеством долларов и количеством центов. Переменная-член, должна содержать значение процентной ставки, представленное в виде дроби, а не в виде процента (в частности, значение процентной ставки 4,3 % нужно представить как значение `0.043` типа `double`). Значение, соответствующее количеству месяцев до наступления срока выплаты, будет храниться так же, как и в программе из листинга 6.1.

4. Определите класс для типа CounterType. Объект этого типа используется для подсчета, так что в нем хранится значение счетчика — неотрицательное целое число. Включите в этот класс конструктор, используемый по умолчанию, который устанавливал бы значение счетчика в нуль, и конструктор с одним аргументом, который присваивал бы счетчику значение этого аргумента. Одна из функций-членов класса должна увеличивать значение счетчика на единицу, а вторая — уменьшать его на единицу. Ни одна из функций-членов не должна допускать, чтобы значение счетчика стало отрицательным. Кроме того, включите в класс такие функции-члены: для возврата текущего значения счетчика и для вывода значения счетчика в заданный поток. Последняя должна иметь один формальный параметр типа ostream для выходного потока. Включите определение этого класса в тестовую программу.

5. Определите класс Month как абстрактный тип данных для информации о месяце. Он должен содержать одну переменную-член типа int, представляющую месяц (1 для января, 2 для февраля и т. д.). Помимо этого, в его состав должны входить такие элементы: конструктор, устанавливающий месяц по первым трем буквам его названия, которые заданы в трех аргументах; конструктор, устанавливающий месяц по его целочисленному номеру, который задан в аргументе (1 для января, 2 для февраля и т. д.); используемый по умолчанию конструктор; функция-член input, считывающая порядковый номер месяца, и функция-член input, считывающая первые три буквы названия месяца; функция-член output, выводящая порядковый номер месяца, и функция-член output, выводящая первые три буквы названия месяца; функция-член, возвращающая следующий месяц как значение типа Month. Функции-члены input и output имеют по одному формальному параметру типа потока. Включите определение этого класса в тестовую программу.

6. Переопределите реализацию класса Month, описанного в предыдущем проекте. На этот раз месяц должен быть реализован в виде трех переменных-членов типа char, в которых хранятся первые три буквы названия месяца. Включите определение данного класса в тестовую программу.

7. (Перед реализацией этого проекта нужно осуществить два предыдущих.) Перепишите программу, приведенную в листинге 6.3, с использованием определенного в проекте 5 или 6 класса Month в качестве типа переменной-члена, хранящей значение месяца. Переопределите функцию-член output так, чтобы у нее был один формальный параметр типа ostream для выходного потока. Модифицируйте программу таким образом, чтобы вся информация, выводимая на экран, помещалась также в файл. Это означает, что каждый оператор вывода должен повторяться дважды: один раз для аргумента cout и второй раз для потокового аргумента. Ввод по-прежнему должен выполняться с клавиатуры. В файл направляются только выходные данные.

8. Напишите класс для рациональных чисел. Воспользуйтесь функциями-членами add, sub, mul, div и less, выполняющими соответствующие операции +, -, *, / и <. Например, операция сложения a + b с помощью этого класса осуществляется как a.add(b), а операция сравнения a < b производится как a.less(b).

Рациональные числа — это числа, которые можно представить в виде дроби с целочисленными числителем и знаменателем, например 1/2, 2/3, 15/32, 65/4, 16/5. В вашем классе рациональные числа должны быть представлены парой значений типа `int` — `numerator` (числитель) и `denominator` (знаменатель).

Для любого абстрактного типа данных необходим конструктор, который создает объекты, представляющие допустимые значения этого типа. В нашем случае конструктор должен создавать объекты на основе пары значений типа `int`. Поскольку любое значение этого типа является рациональным числом (его можно представить как это же число, деленное на 1), необходим еще один конструктор с одним целочисленным параметром.

Включите в класс функции-члены `input` и `output`. Первая из них принимает аргумент типа `istream` и вводит рациональное число в форме числитель/знаменатель с клавиатуры или из файла. Вторая принимает аргумент типа `ostream` и выводит рациональное число в форме числитель/знаменатель на экран или в файл.

Функции-члены `add`, `sub`, `mul` и `div` должны возвращать рациональные значения. Функция-член `less` возвращает значение типа `bool`. Кроме того, включите в класс функцию-член `neg`, не имеющую параметров и возвращающую вызывающий объект, взятый со знаком минус. Напишите функцию `main` для тестирования реализации вашего класса. При определении функций вам будут полезны следующие формулы:

```
a/b + c/d = (a*d + b*c) / (b*d)
a/b - c/d = (a*d - b*c) / (b*d)
(a/b) * (c/d) = (a*c) / (b*d)
(a/b) / (c/d) = (a*d) / (c*b)
-(a/b) = (-a/b)
(a/b) < (c/d) означает (a*d) < (c*b)
(a/b) == (c/d) означает (a*d) == (c*b)
```

Числитель может иметь любой знак, а знаменатель должен быть только положительным.

Мы еще вернемся к этой задаче в главе 8, где рассказывается о перегрузке операторов, позволяющей значительно облегчить ее решение.

Глава 7 Поток управления программы

Если вы оказались на распутье, придется сделать выбор.
Приписывается Йоги Берра

Порядок выполнения операторов в программе называется *потоком управления*. Вам уже знакомы три оператора, с помощью которых он задается: `if...else`, `while` и `do...while`. В данной главе рассматриваются некоторые новые способы использования этих операторов и рассказывается о двух новых операторах управления потоком: `switch` и `for`. Кроме того, поскольку действие операторов `if...else`, `while` и `do...while` основано на проверке логических выражений, мы подробнее поговорим о логических выражениях.

7.1. Использование логических выражений

— И наоборот, — подхватил Траляля.
— Если бы это было так, это бы еще ничего,
а если бы ничего, оно бы так и было,
но так как это не так, так оно и не этак!
Такова логика вещей!
Льюис Кэрролл

Вычисление логических выражений

Логическое выражение — это выражение, которое может быть истинным или ложным. До сих пор мы употребляли логические выражения для проверки условий в операторе `if...else`, а также применяли их в качестве управляющих выражений в циклах (таких, как `while`), но этим их использование не ограничивается.

Для хранения результатов вычисления логических выражений, то есть значений `true` и `false`, в C++ предусмотрен специальный тип данных, называемый `bool`.

Логическое выражение вычисляется точно так же, как арифметическое. Различаются они только тем, что во втором используются арифметические операторы +, -, * и /, и его результатом является число, тогда как в логическом выражении применяются операторы сравнения, например == и <, а также логические операторы, такие, как &&, || и !. Операторы =, !=, <, <= и т. п. выполняются над парой значений любого встроенного типа и возвращают логическое значение true или false. Разобравшись, как вычисляются логические выражения, вы сможете составлять выражения любой сложности и использовать их в разных ситуациях, в том числе в качестве возвращаемых функциями значений.

Сначала рассмотрим процесс вычисления арифметического выражения, поскольку лежащий в его основе принцип полностью применим и к логическим выражениям. Возьмем следующее арифметическое выражение:

```
(x + 1) * (x + 3)
```

Предположим, что переменная x содержит значение 2. Для того чтобы вычислить данное выражение, нужно вычислив две суммы, получить значения 3 и 5, а затем, перемножив их, вычислить окончательный результат — значение 15. Обратите внимание, что умножение выражений (x + 1) и (x + 3) в данном случае не производится, вместо этого перемножаются их значения. Иными словами, операндами умножения являются значения 3, а не (x + 1), и 5, а не (x + 3).

Точно так же обрабатываются и логические выражения. Сначала вычисляются вложенные выражения, в результате чего получается значение true или false. Эти отдельные значения true или false комбинируются в соответствии с правилами, приведенными в таблицах на рис. 7.1.

AND

Выражение_1	Выражение_2	Выражение_1 && Выражение_2
true	true	true
true	false	false
false	true	false
false	false	false

OR

Выражение_1	Выражение_2	Выражение_1 || Выражение_2
true	true	true
true	false	true
false	true	true
false	false	false

NOT

Выражение	!(Выражение)
true	false
false	true

Рис. 7.1. Таблицы истинности

В качестве примера рассмотрим логическое выражение

```
!((y < 3) || (y > 7))
```

которое может использоваться для управления оператором `if...else` или `while`. Предположим, что значение переменной `y` равно 8. Тогда результатом вычисления выражения `(y < 3)` будет `false`, а результатом вычисления выражения `(y > 7)` — `true`, так что приведенное выше выражение с данным значением переменной `y` эквивалентно следующему:

```
!(false || true)
```

Обратившись к рис. 7.1, найдем значение оператора `||` (операция `OR` — логическое «ИЛИ») и увидим, что выражение в скобках возвращает `true`. Поэтому все выражение эквивалентно такому:

```
!(true)
```

Отыщем в таблице значение оператора `!` (операция `NOT` —логическое «НЕ»), и увидим, что значение выражения `!(true)` равно `false`, а значит значением исходного выражения является `false`.

Почти во всех примерах, рассмотренных нами до сих пор, имелся полный набор скобок, показывающих, к каким выражениям относится каждый оператор `&&`, `||` и `!`. Однако это не является обязательным условием. При отсутствии скобок операторы выполняются в следующем порядке: сначала `!`, далее операторы сравнения (такие, как `>`), затем `&&` и только потом `||`. Но для облегчения чтения и понимания логических выражений лучше помещать в них достаточное количество скобок. Скобки безопасно опускать в простых последовательностях операндов, объединяемых при помощи оператора `&&` или `||` (но только в том случае, если в выражении используется один из операторов, а не оба). Например, следующее выражение вполне приемлемо и с точки зрения компилятора C++, и с точки зрения читабельности:

```
(temperature > 90) && (humidity > 0.90) && (pool_gate == OPEN)
```

Поскольку операторы сравнения `>` и `==` выполняются перед оператором `&&`, в приведенном выше выражении можно опустить скобки, и его значение от этого не изменится. Но использование скобок, тем не менее, значительно облегчает его чтение.

Если скобки в выражении опущены, компьютер группирует элементы в соответствии с *правилами старшинства операций*. Некоторые из этих правил, принятых в языке C++, приведены на рис. 7.2. Если одна операция всегда выполняется перед другой, говорят, что она имеет более высокий *приоритет*. Бинарные операторы с равным приоритетом выполняются слева направо в том порядке, в котором они заданы в выражении. Унарные операторы с равным приоритетом выполняются справа налево. Полный набор правил старшинства операций приведен в приложении 2.

Обратите внимание, что правила старшинства операций определяют порядок выполнения как арифметических операторов (таких, как `+` или `*`), так и логических (например, `&&` или `||`). Дело в том, что многие выражения включают операторы обоих типов, как в следующем примере:

```
(x + 1) > 2 || (x + 1) < -3
```

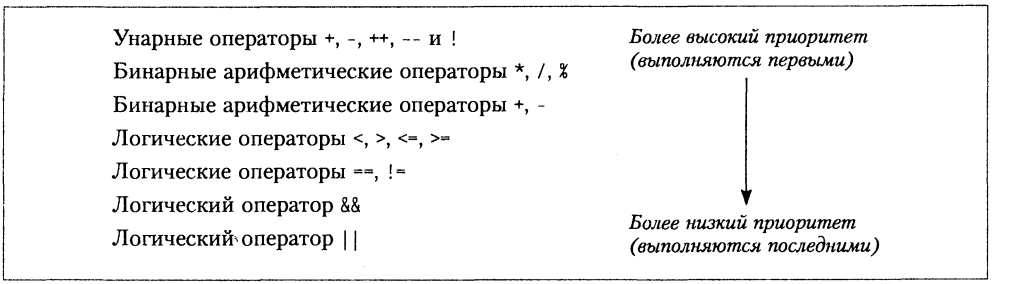


Рис. 7.2. Правила старшинства операций

Обратившись к правилам старшинства операций, приведенным на рис. 7.2, вы увидите, что это выражение эквивалентно следующему:

```
((x + 1) > 2) || ((x + 1) < -3)
```

поскольку операторы > и < имеют более высокий приоритет, чем оператор ||. Фактически в этом выражении можно опустить вообще все скобки, однако читать его в таком случае будет неудобно. Поэтому мы не советуем так поступать, но с чисто учебной целью разберем процесс его интерпретации вот в таком виде:

```
x + 1 > 2 || x + 1 < -3
```

Правила старшинства операций гласят, что сначала должен быть применен унарный минус, далее выполнено сложение, затем сравнение с помощью операторов > и <, а потом оператор ||.

Приведенное в настоящем разделе общее описание процесса вычисления логических выражений в целом верно, но следует добавить, что в C++ применяется алгоритм его ускорения. Просмотрев приведенные на рис. 7.1 таблицы истинности логических операторов || и &&, вы могли заметить, что во многих случаях достаточно вычислить только один из операндов. В качестве примера рассмотрим такое выражение:

```
(x >= 0) && (y > 1)
```

Если `x` отрицательно, выражение `(x >= 0)` равно `false`. Как видно из таблицы на рис. 7.1, оператор && возвращает `false`, когда один из его операндов равен `false`, и в этом случае значение второго операнда не важно. Поэтому, если известно, что первое выражение равно `false`, нет смысла вычислять второе. Ситуация с оператором || аналогична. Если первое из выражений, объединенных данным оператором, имеет значение `true`, оператор возвращает `true` независимо от значения второго выражения. Это используется в C++ для ускорения вычисления логических выражений. Сначала вычисляется первое из выражений, объединенных оператором || или &&, и в случае получения достаточной информации для определения значения операции, второе выражение не вычисляется. Этот метод называется *сокращенным вычислением выражений.*

В некоторых других языках программирования применяется полное вычисление выражений. То есть при выполнении оператора || или && сначала вычисляются все составляющие выражения, а затем — результат сложного выражения.

Применение и той, и другой схемы дает один и тот же результат. Так зачем же знать, что в C++ используется именно сокращенная схема? В большинстве случаев это не имеет значения, но иногда может быть очень важно. Если оба подвыражения, объединенные оператором || или &&, содержат значения, оба метода возвращают один и тот же результат. Однако если значение второго подвыражения не определено, сокращенное вычисление может быть очень полезно. Давайте рассмотрим пример:

```
if ((family != 0) && ((kids/family) >= 2))
    cout << "Each child may have two pieces!";
```

Если значение переменной `family` не равно нулю, оператор выполняется полностью. Однако в случае его равенства нулю процесс сократится, поскольку значением выражения `(family != 0)` будет `false`, и отпадет необходимость вычислять второе выражение. C++ сразу определяет, что все выражение в операторе `if` равно `false`, и не вычисляя выражение `((kids/family) >= 2)`, переходит к следующему оператору программы. Это предотвращает ошибку времени выполнения, поскольку вычисление второго выражения потребовало бы деления на нуль.

Иногда C++ обращается с целыми числами так, как с логическими выражениями. В частности, он преобразует значение `1` в `true`, а значение `0` в `false`. Точнее, компилятор преобразует в `false` любое число, отличное от нуля. Если при написании логических выражений быть внимательным и не допускать ошибок, в таком преобразовании нет ничего сложного. Но при отладке программы полезно все же иметь в виду, что компилятор способен без проблем выполнять логические операции над целыми числами.

> **Логические значения — это true и false**
>
> В C++ логическое выражение возвращает значение типа `bool`, то есть `true`, когда это выражение истинно, и `false` в противном случае.

Ловушка: логические выражения преобразуются в значения типа int

Предположим, вы хотите использовать логическое выражение в операторе `if...else` и необходимо, чтобы оно имело значение `true`, пока не истечет некоторое заданное время (скажем, для какой-нибудь игры или реального процесса). Сформулируем это более точно: в операторе `if...else` необходимо логическое выражение, равное `true`, если значение переменной `time` типа `int` не больше значения переменной `limit`. Этот оператор можно записать так:

```
if (!time > limit)   // Не подходит для нашей задачи.
    операторы
else
    операторы
```

Вроде бы все звучит правильно: «Если неверно, что значение переменной `time` бо. ьше значения переменной `limit`». На самом же деле с точки зрения логики

программы выражение составлено неверно, хотя компилятор не выдаст сообщения об ошибке. А ошибка заключается в том, что здесь не учтены приоритеты операторов ! и >. Компилятор, который будет строго придерживаться правил старшинства операций, интерпретирует выражение, приведенное выше, так:

```
(!time) > limit
```

что является бессмыслицей. Если значение переменной time равно, скажем, 36, то что может означать выражение (!time)? Что такое «не 36»? Однако в C++ любое ненулевое целое число преобразуется в значение true, а нуль в значение false, поэтому !36 будет интерпретировано как false. А поскольку с помощью оператора > сравниваются целые числа, значение false снова преобразуется в int, то есть в 0.

Таким образом, приведенный выше код будет работать, но только совсем не так, как нам требуется. Например, если переменная time равна 36, а переменная limit — 60, нам нужно, чтобы выражение в операторе if...else вернуло true, поскольку неверно, что time > limit. Но приведенное выше логическое выражение вычисляется так: (!time) преобразуется в 0, а все логическое выражение превращается в

```
0 > limit
```

что эквивалентно 0 > 60, то есть false. Итак, результатом вычисления логического выражения оказалось false, тогда как нам нужно было true.

Существует два способа исправить дело. Первый из них заключается в правильном применении оператора !. Поскольку его приоритет выше, чем у оператора сравнения, оператор сравнения нужно просто заключить в скобки, вот так:

```
if (!(time > limit))
    операторы
else
    операторы
```

Второй способ решения проблемы — вообще избавиться от оператора !. Так, следующее выражение не только правильное, но еще и более читабельное:

```
if (time <= limit)
    операторы
else
    операторы
```

В большинстве выражений можно обойтись без оператора отрицания, и многие программисты настаивают на том, что его использования следует по возможности избегать. Они считают, что выражение с оператором отрицания обычно вносит некоторую неопределенность, а кроме того, является недостаточно читабельным.

Упражнения для самопроверки

1. Определите значение каждого из следующих логических выражений при условии, что переменная count равна 0, а переменная limit равна 10. Ответом должно быть одно из двух значений: true или false.

   а) `(count == 0) && (limit < 20);`

   б) `count == 0 && limit < 20;`

в) `(limit > 20) || (count < 5);`

г) `!(count == 12);`

д) `(count == 1) && (x < y);`

е) `(count < 10) || (x < y);`

ж) `!(((count < 10) || (x < y)) && (count >= 0));`

з) `((limit/count) > 7) || (limit < 20);`

и) `(limit < 20) || ((limit/count) > 7);`

к) `((limit/count) > 7) && (limit < 0);`

л) `(limit < 0) && ((limit/count) > 7);`

м) `(5 && 7) + (!6).`

2. Назовите два вида операторов C++, изменяющих порядок выполнения операций программы. Приведите несколько примеров.

3. В школьном курсе алгебры числовые интервалы задаются так:

   ```
   2 < x < 3
   ```

   В C++ это выражение имеет совсем иное значение. Объясните, почему, и приведите правильное логическое выражение C++, определяющее, что значение переменой `x` находится в диапазоне от 2 до 3.

4. Происходит ли деление на нуль при выполнении следующего кода:

   ```
   j = -1;
   if ((j > 0) && (1/(j+1) > 10))
   cout << i << endl;
   ```

Функции, возвращающие логические значения

Функция может возвращать значение типа `bool`. Такую функцию используют в логических выражениях для управления оператором `if...else`, управления циклом и везде, где допускается применение логических выражений.

Логические функции часто помогают сделать программу более структурированной и читабельной. Сложные выражения затрудняют восприятие программного кода, поэтому будет лучше, если присвоить подобному выражению информативное имя (то есть определить для него отдельную функцию) и подставлять его вместо самого выражения в другие логические выражения, условия операторов ветвления и циклов, а также везде, где можно использовать логические выражения. Так, оператор

```
if (((rate >= 10) && (rate < 20)) || (rate == 0))
{
    ...
}
```

будет читаться гораздо легче, если переписать его следующим образом:

```
if (appropriate(rate))
{
    ...
}
```

при условии определения функции

```
bool appropriate(int rate)
{
    return (((rate >= 10) && (rate < 20)) || (rate == 0));
}
```

Упражнения для самопроверки

5. Напишите определение функции in_order, принимающей три аргумента типа int. Эта функция возвращает true, если аргументы расположены по возрастанию значений; в противном случае она возвращает false. Например, вызовы функций in_order(1, 2, 3) и in_order(1, 2, 2) возвращают значение true, а вызов in_order(1, 3, 2) возвращает false.
6. Напишите определение функции even, принимающей один аргумент типа int и возвращающей значение типа bool. Функция возвращает true, если ее аргумент является четным числом, и false в противном случае.
7. Напишите определение функции is_digit, принимающей один аргумент типа char и возвращающей значение типа bool. Эта функция возвращает true, если ее аргумент является цифрой, и false в противном случае.
8. Напишите определение функции is_root_of, принимающей два аргумента типа int и возвращающей значение типа bool. Данная функция возвращает true, если ее первый аргумент равен квадратному корню из второго, и false в противном случае.

Перечисления (факультативный материал)

Перечисление — это тип данных, значения которого определены как список констант типа int. Перечисление очень напоминает список объявленных констант.

В определении перечисления можно использовать любые значения типа int и любое количество констант. Например, следующее перечисление определяет константы для продолжительности каждого месяца:

```
enum MonthLength {JAN_LENGTH = 31, FEB_LENGTH = 28,
    MAR_LENGTH = 31, APR_LENGTH = 30, MAY_LENGTH = 31,
    JUN_LENGTH = 30, JUL_LENGTH = 31, AUG_LENGTH = 31,
    SEP_LENGTH = 30, OCT_LENGTH = 31, NOV_LENGTH = 30,
    DEC_LENGTH = 31};
```

Как видно из этого примера, две или более константы, входящие в состав перечисления, могут иметь одно и то же значение.

Если не указаны никакие числовые значения, идентификаторам в определении перечисления присваиваются последовательные значения, начиная с 0. Так, определение типа

```
enum Direction { NORTH = 0, SOUTH = 1, EAST = 2, WEST = 3};
```

эквивалентно следующему:

```
enum Direction { NORTH, SOUTH, EAST, WEST };
```

Эта вторая форма объявления перечисления без явно заданного списка целочисленных значений обычно используется в тех случаях, когда требуется только список имен и при этом не важны значения, которые с ними связаны.

Если инициализировать часть констант перечисления, например:

```
enum MyEnum { ONE = 17, TWO, THREE, FOUR = -3, FIVE };
```

константе ONE будет присвоено значение 17, TWO — следующее большее целочисленное значение, то есть 18, THREE — значение 19, FOUR — значение -3 и FIVE — следующее большее целочисленное значение, -2.

Первая константа перечисления по умолчанию получает значение 0, а значение каждой следующей константы на единицу больше значения предыдущей.

7.2. Многонаправленное ветвление

— Скажите, пожалуйста, куда мне отсюда идти?
—Это во многом зависит от того, куда ты хочешь прийти, — ответил Кот.

Льюис Кэрролл

Любая программная конструкция, в которой осуществляется выбор одного из множества альтернативных действий, называется *механизмом ветвления*. Вы уже знакомы с оператором if...else, который осуществляет выбор из двух возможных действий. В этом разделе мы поговорим о методах выбора из более чем двух возможностей.

Вложенные операторы if...else

Как вы уже знаете, операторы if...else и if являются составными, то есть содержат другие операторы программы. До сих пор в наших примерах они включали только простые одиночные операторы или линейные последовательности операторов в фигурных скобках. Однако на самом деле в них могут содержаться любые операторы, включая if...else, while и do...while. На рис. 7.3 показан оператор с тремя уровнями вложенности, обозначенными прямоугольниками, — два оператора cout вложены в оператор if...else, а тот, в свою очередь, вложен в оператор if.

При записи вложенных операторов каждый их уровень обычно сдвигают вправо, чтобы наглядно показать структуру программы. Так на рис. 7.3 для трех уровней вложенности используется соответствующее количество отступов. Обратите внимание, что поскольку оба оператора cout расположены на одном уровне вложенности, величина их отступов одинакова. Далее в этой главе будут рассмотрены

случаи, когда используется иная схема отступов, но обычно рекомендуется каждый уровень вложенности отмечать дополнительным отступом.

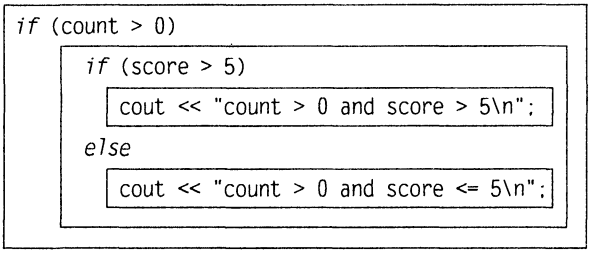


Рис. 7.3. Оператор if...else, вложенный в оператор if

Совет программисту: используйте скобки во вложенных операторах

Предположим, вы хотите написать оператор `if...else` для использования в компьютерной системе гоночного автомобиля. Эта часть программы предупреждает водителя о снижении уровня горючего в баке, а также подсказывает ему, что следует пропускать пит-стопы (остановки для дозаправки), когда бак полон. В остальных случаях программа ничего не выводит, чтобы не отвлекать водителя. Мы разработали следующий псевдокод.

```
Если бак заполнен меньше, чем на 3/4, то:
    проверить, заполнен ли он меньше чем на 1/4,
    и если да, выдать предупреждение о малом количестве топлива.
В противном случае (то есть, если бак заполнен более чем на 3/4):
    вывести указание не останавливаться на дозаправку.
```

Если невнимательно реализовать этот псевдокод, результат может быть таким:

```
if (fuel_gauge_reading < 0.75)
    if (fuel_gauge_reading < 0.25)
        cout << "Fuel very low. Caution!\n";
else
    cout << "Fuel over 3/4. Don't stop now!\n";
```

На первый взгляд приведенная реализация может показаться нормальной, и с точки зрения языка C++ в ней действительно все в порядке: компилятор ее примет и при выполнении не будет сгенерировано сообщений об ошибках. Но она не соответствует нашему псевдокоду. Обратите внимание, что здесь содержится два условия `if` и только одна ветвь `else`. Компилятор должен решить, какому из условий `if` данная ветвь соответствует. Мы красиво расположили этот многоуровневый оператор, показав отступами, куда относится ветвь `else`, но проблема в том, что компилятор не будет смотреть на отступы. Для него приведенный выше оператор ничем не отличается от следующего, в котором отступы расположены иначе:

```
if (fuel_gauge_reading < 0.75)
    if (fuel_gauge_reading < 0.25)
        cout << "Fuel very low. Caution!\n";
    else
        cout << "Fuel over 3/4. Don't stop now!\n";
```

К сожалению, компилятор интерпретирует наш код именно таким способом и поставит ветвь else в соответствие второму условию if. Эту ситуацию иногда называют *проблемой повисшего else*, а из программы листинга 7.1 видно, в чем данная проблема заключается.

Компилятор всегда соотносит ключевое слово else с ближайшим предшествующим ключевому словом if, не имеющим ветви else. Но нет никакой необходимости подчиняться этому правилу. Всегда имеется возможность указать компилятору, чего именно вы хотите, так как вы являетесь автором программы, и она должна делать то, что требуется вам. Каким же образом показать компилятору настоящую структуру вложенных операторов if...else? Да очень просто: воспользуйтесь фигурными скобками, которые во вложенных операторах выполняют ту же функцию, что круглые скобки в выражениях. Они указывают компилятору, как следует группировать элементы кода, чтобы он соответствовал поставленной задаче, а не встроенным правилам. Во избежание проблем и для облегчения чтения программы заключайте вложенный код операторов if...else в фигурные скобки, как в первом операторе if...else листинга 7.1.

Для простейших операторов, таких как единственное присваивание или единственный оператор cout, фигурные скобки можно опустить. Например, в листинге 7.1 вложенный оператор (в первом операторе if...else)

```
cout << "Fuel over 3/4. Don't stop now!\n";
```

можно не заключать в фигурные скобки. Однако даже в таких простых случаях использование скобок иногда облегчает чтение программы. Некоторые программисты настаивают на том, чтобы заключать в фигурные скобки даже простейшие операторы, входящие в состав операторов if...else.

Листинг 7.1. Важность использования фигурных скобок

```
// Программа иллюстрирует важность использования фигурных скобок в операторах if...else.
#include <iostream>
using namespace std;
int main()
{
    double fuel_gauge_reading;

    cout << "Enter fuel gauge reading: ";
    cin >> fuel_gauge_reading;

    cout << "First with braces:\n";
    if (fuel_gauge_reading < 0.75)
{
        if (fuel_gauge_reading < 0.25)
            cout << "Fuel very low. Caution!\n";
}
else
{
        cout << "Fuel over 3/4. Don't stop now!\n";
}

    cout << "Now without braces:\n";
    if (fuel_gauge_reading < 0.75)
        if (fuel_gauge_reading < 0.25)
```

```
        cout << "Fuel very low. Caution!\n";
    else
        cout << "Fuel over 3/4. Don't stop now!\n";

    return 0;
}
```

Пример диалога 1

```
Enter fuel gauge reading: 0.1
First with braces:
Fuel very low. Caution!
Now without braces:
Fuel very low. Caution!
```

Пример диалога 2

```
Enter fuel gauge reading: 0.5
First with braces:
Now without braces:
Fuel over 3/4. Don't stop now!
```

Многонаправленные операторы if...else

Оператор `if...else` осуществляет двунаправленное ветвление, позволяя программе выбрать одно из двух возможных действий. Однако часто ей требуется выбор из большего количества действий, и тогда можно воспользоваться вложенными операторами `if...else`. Предположим, нам нужно разработать игровую программу, в которой пользователь должен угадать число. Это число хранится в переменной `number`, а ответ игрока является значением переменной `guess`. Если после каждой попытки мы намерены выводить подсказку, алгоритм программы может быть описан следующим псевдокодом:

```
Вывести "Too high.", если guess > number.
Вывести "Too low.", если guess < number.
Вывести "Correct!", если guess == number.
```

Когда ветвление определяется как список взаимоисключающих условий и соответствующих действий, его можно реализовать с помощью вложенных операторов `if...else`. Например, приведенный выше псевдокод реализуется так:

```
if (guess > number)
    cout << "Too high.";
else if (guess < number)
    cout << "Too low.";
else if (guess == number)
    cout << "Correct!";
```

Здесь расположение кода отличается от предложенной выше схемы. Если бы мы следовали описанным ранее правилам, этот код выглядел бы так:

```
if (guess > number)
    cout << "Too high.";
else
    if (guess < number)
        cout << "Too low.";
    else
```

```
    if (guess == number)
        cout << "Correct!";
```

Но это один из тех редких случаев, когда не нужно следовать общим правилам расположения кода и отступов во вложенных операторах. Выравнивая все ключевые слова `else`, вы выравниваете и пары условие-действие так, что программа отражает логику операций.

Поскольку условия всех операторов взаимоисключающие, последнее из них излишне, и его можно не задавать, хотя иногда вместо него лучше поместить хотя бы комментарий:

```
if (guess > number)
    cout << "Too high.";
else if (guess < number)
    cout << "Too low.";
else   // (guess == number)
    cout << "Correct!";
```

Эта форма оператора `if...else` с множественным ветвлением может использоваться и в тех случаях, когда условия не являются взаимоисключающими. Компилятор в любом случае будет оценивать условия по порядку, и найдя первое удовлетворяющееся условие, выполнит соответствующее ему действие. Если ни одно из условий не истинно, не выполняется ни одно из действий. Если же оператор завершается ветвью `else` без последнего условия `if`, последний оператор выполняется, когда все условия оказываются ложными.

Многонаправленные операторы if...else

Синтаксис

```
if (логическое_выражение_1)
    оператор_1
else if (логическое_выражение_2)
    оператор_2
        .
        .
        .
else if (логическое_выражение_n)
    оператор_n
else
    оператор_для _всех_остальных_случаев
```

Пример

```
if ((temperature < -10) && (day == SUNDAY))
    cout << "Stay home.";
else if (temperature < -10)   // day != SUNDAY
    cout << "Stay home, but call work.";
else if (temperature <= 0)    // temperature >= -10
    cout << "Dress warm.";
else   // temperature > 0
    cout << "Work hard and play hard.";
```

Логические выражения проверяются по порядку, пока не встретится первое истинное выражение, и тогда выполняется соответствующий оператор. Если ни одно из логических выражений не истинно, выполняется *оператор_для_всех_остальных_случаев*.

Пример: подоходный налог штата

В программе листинга 7.2 приведено определение функции, в которой используется многонаправленный оператор if...else. Данная функция принимает один аргумент, значение которого представляет сумму дохода налогоплательщика, подлежащую обложению налогом (в долларах), округленное до целого числа, и вычисляет соответствующий подоходный налог. В данном штате принята следующая схема вычисления налога.

1. Первые $15 000 налогом не облагаются.
2. На каждый доллар от $15 001 до $25 000 начисляется 5 % налога.
3. На каждый доллар свыше $25 000 начисляется 10 % налога.

В функции, определенной в листинге 7.2, выполняется оператор if...else с одним действием для каждого условия. Второе условие в этом операторе сложнее. Компьютер не перейдет к нему, не проверив первое условие и не убедившись, что оно ложно. Поэтому к моменту проверки второго условия уже известно, что значение переменной net_income больше 15000. Следовательно, строку

```
else if ((net_income > 15000) && (net_income <= 25000))
```

можно заменить строкой

```
else if (net_income <= 25000)
```

Листинг 7.2. Многонаправленный оператор if...else

```
// Программа для вычисления подоходного налога.
#include <iostream>
using namespace std;


double tax(int net_income);
// Предусловие: формальный параметр net_income представляет
// значение облагаемого налогом дохода, округленное до целого числа.
// Возвращает сумму подлежащего выплате налога, вычисляемую
// по следующей формуле:
// первые $15 000 налогом не облагаются; 5 % начисляется
// на сумму от $15 001 до $25 000 и еще 10 % – на доход свыше $25 000.


int main()
{
    int net_income;
    double tax_bill;

    cout << "Enter net income (rounded to whole dollars) $";
    cin >> net_income;

    tax_bill = tax(net_income);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Net income = $" << net_income << endl
         << "Tax bill = $" << tax_bill << endl;
```

продолжение ⇨

Листинг 7.2 (*продолжение*)

```
    return 0;
}


double tax(int net_income)
{
    double five_percent_tax, ten_percent_tax;

if (net_income <= 15000)
        return 0;
    else if ((net_income > 15000) && (net_income <= 25000))
        // Возвращает 5 % суммы свыше $15 000
        return (0.05*(net_income - 15000));
    else // net_income > $25,000
    {
        // five_percent_tax = 5 % дохода от $15 000 до $25 000.
        five_percent_tax = 0.05*10000;
        // ten_percent_tax = 10% дохода свыше $25,000.
        ten_percent_tax = 0.10*(net_income - 25000);
        return (five_percent_tax + ten_percent_tax);
    }
}
```

Пример диалога

```
Enter net income (rounded to whole dollars) $25100
Net income = $25100
Tax bill = $510.00
```

Упражнения для самопроверки

9. Что выведет код

```
int x = 2;
cout << "Start\n";
if (x <= 3)
    if (x != 0)
        cout << "Hello from the second if.\n";
    else
        cout << "Hello from the else.\n";
cout << "End\n";

cout << "Start again\n";
if (x > 3)
    if (x != 0)
        cout << "Hello from the second if.\n";
    else
        cout << "Hello from the else.\n";
cout << "End again\n";
```

при выполнении в составе полной программы?

10. Что выведет код

```
int extra = 2;
    if (extra < 0)
        cout << "small";
    else if (extra == 0)
```

```
        cout << "medium";
    else
        cout << "large";
```

при выполнении в составе полной программы?

11. Что выведет код, приведенный в упражнении 10, если заменить оператор присваивания следующим:

```
int extra = -37;
```

12. Что выведет код, приведенный в упражнении 10, если заменить оператор присваивания следующим:

```
int extra = 0;
```

13. Что выведет код

```
int x = 200;
cout << "Start\n";
if (x < 100)
    cout << "First Output.\n";
else if (x > 10)
    cout << "Second Output.\n";
else
    cout << "Third Output.\n";
cout << "End\n";
```

при выполнении в составе полной программы?

14. Что выведет код, приведенный в упражнении 13, если заменить выражение (x > 10) выражением (x > 100)?

15. Что выведет код

```
int x = SOME_CONSTANT;
cout << "Start\n";
if (x < 100)
    cout << "First Output.\n";
else if (x > 100)
    cout << "Second Output.\n";
else
    cout << x << endl;
cout << "End\n";
```

при выполнении в составе полной программы? Здесь SOME_CONSTANT — константа типа int, которой вы присвоили значение 100.

16. Напишите оператор if...else, относящий значение переменной n типа int к одной из следующих категорий:

$n < 0$, $0 \le n \le 100$, $n > 100$

и выводящий соответствующее сообщение.

17. Что выведет следующий поток cout, при наличии в программе такого объявления:

```
enum Direction {N, S, E, W};
//...
cout << W << " " << E << " " << S << " " << N << endl;
```

18. Что выведет следующий поток cout при наличии в программе такого объявления:

```
enum Direction {N = 5, S = 7, E = 1, W};
//...
cout << W << " " << E << " " << S << " " << N << endl;
```

Оператор switch

Вы уже знаете, как организовать многонаправленное ветвление с помощью операторов if...else. В языке C++ имеется еще один оператор, позволяющий решить такую задачу. Это оператор switch, специально предназначенный для реализации многонаправленного ветвления. Пример его применения приведен в листинге 7.3. Здесь содержатся четыре ветви для обработки допустимых входных значений и пятая для обработки недопустимых значений. Программа запрашивает ввод оценки успеваемости и выводит соответствующий комментарий. Каждой из оценок 'A', 'B' и 'C' соответствует отдельная ветвь программы. Оценкам 'D' и 'F' соответствует общая ветвь. Если значением переменной grade, в которой хранится введенная оценка, является любой символ, отличный от 'A', 'B', 'C', 'D' или 'F', то выполняется оператор cout, расположенный после ключевого слова default.

Синтаксис и оптимальная схема расположения оператора switch приведены в программе листинга 7.3 и во врезке «Оператор switch».

Листинг 7.3. Оператор switch

```
// Программа, демонстрирующая использование оператора switch.
#include <iostream>
using namespace std;

int main()
{
    char grade;

    cout << "Enter your midterm grade and press return: ";
    cin >> grade;

switch (grade)
    {
        case 'A':
            cout << "Excellent. "
                 << "You need not take the final.\n";
            break;
        case 'B':
            cout << "Very good. ";
            grade = 'A';
            cout << "Your midterm grade is now."
                 << grade << endl;
            break;
        case 'C':
            cout << "Passing.\n";
            break;
        case 'D':
        case 'F':
            cout << "Not good. "
```

```
                << "Go study.\n";
            break;
        default:
            cout << "That is not a possible grade.\n";
    }

    cout << "End of program.\n";
    return 0;
}
```

Пример диалога 1

```
Enter your midterm grade and press return: A
Excellent. You need not take the final.
End of program.
```

Пример диалога 2

```
Enter your midterm grade and press return: B
Very good. Your midterm grade is now A
End of program.
```

Пример диалога 3

```
Enter your midterm grade and press return: D
Not good. Go study.
End of program.
```

Пример диалога 4

```
Enter your midterm grade and press return: E
That is not a possible grade.
End of program.
```

Оператор switch определяет несколько ветвей программы, из которых выполняется только одна. Выбор осуществляется на основе управляющего выражения, заданного в скобках после ключевого слова switch. В примере листинга 7.3 это выражение типа char. В общем случае управляющее выражение оператора switch должно возвращать либо значение типа bool, либо константу типа enum, либо значение одного из целочисленных типов, либо символ. При выполнении оператора switch вычисляется управляющее выражение, а затем компьютер просматривает значения констант, заданных после ключевых слов case. Встретив константу, значение которой совпадает со значением управляющего выражения, он выполняет код данной ветви case. Например, если проверка управляющего выражения возвращает 'B', C++ ищет приведенную ниже строку и выполняет следующий за ней оператор:

```
case 'B':
```

Обратите внимание, что за константой следует двоеточие. Учтите, что оператор switch не может содержать две ветви case с одинаковыми константами.

Оператор break состоит из ключевого слова break, за которым следует точка с запятой. Когда компьютер выполняет операторы, расположенные после ключевого слова case, он делает это последовательно до тех пор, пока не встретит оператор break. В этом случае выполнение оператора switch заканчивается. Если опустить

операторы break, после выполнения кода одной ветви case компьютер будет переходить к следующей.

Заметьте, что одному фрагменту исполняемого кода могут соответствовать две ветви case. В операторе switch (см. листинг 7.3) одно и то же действие выполняется для значений 'D' и 'F'. Это свойство оператора switch может использоваться для одинаковой обработки букв верхнего и нижнего регистров. Так, чтобы программа в листинге 7.3 разрешала вводить как 'A', так и 'a', ветвь

```
case 'A':
    cout << "Excellent. "
         << "You need not take the final.\n";
         break;
```

нужно заменить следующим кодом:

```
case 'A':
case 'a':
    cout << "Excellent. "
         << "You need not take the final.\n";
         break;
```

То же самое можно сделать и для всех остальных букв.

Если ни в одной из ветвей case не задана константа, соответствующая значению управляющего выражения, выполняется оператор, следующий за ключевым словом default. Раздел default в операторе switch не обязателен, и если он отсутствует, то в указанном случае работа оператора switch просто завершается. Однако лучше всегда включать в этот оператор раздел default, поскольку, если имеющиеся ветви case охватывают все возможные допустимые ситуации, в этот раздел можно поместить оператор вывода сообщения об ошибке. Именно так мы поступили в программе из листинга 7.3.

Оператор switch

Синтаксис

```
switch (управляющее_выражение)
{
    case константа_1:
        последовательность_операторов_1
        break;
    case константа_2:
        последовательность_операторов_2
        break;
            .
            .
            .
    case константа_n:
        последовательность_операторов_n
        break;
    default:
        последовательность_операторов_по_умолчанию
}
```

Пример

```
int vehicle_class;
cout << "Enter vehicle class: ";
cin >> vehicle_class;
switch (vehicle_class)
{
    case 1:
        cout << "Passenger car.";
        toll = 0.50;
        break;   // Если не поместить этот оператор break, владельцы
                 // легковых машин будут платить $1.50.
    case 2:
        cout << "Bus.";
        toll = 1.50;
        break;
    case 3:
        cout << "Truck.";
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!";
}
```

Ловушка: забытый оператор break в операторе switch

Если забыть расставить в операторе switch операторы break, компилятор не выдаст сообщения об ошибке. Синтаксически все будет правильно, но оператор switch будет делать не то, что планировалось. Рассмотрим пример оператора switch, приведенный во врезке «Оператор switch». Если случайно удалить из него закомментированный оператор break, то в случае, когда переменная vehicle_class содержит значение 1, будет, как и планировалось, выполнен код, следующий за строкой

```
case 1:
```

Однако затем, вместо того чтобы выйти из оператора switch, компьютер выполнит следующую ветвь case. В результате на экран будет выведено противоречивое сообщение, утверждающее, что данный вид транспорта — легковой автомобиль, а затем, что он является автобусом. Более того, переменной toll в итоге будет присвоено значение 1.50, а не 0.50, как планировалось. Начав выполнение кода, соответствующего одной из ветвей case, компьютер не останавливается до тех пор, пока не достигнет оператора break или конца оператора switch.

Использование операторов switch для создания меню

Многонаправленные операторы if...else более гибкие, чем операторы switch, и нет такого оператора switch, для которого нельзя было бы написать эквивалентный оператор if...else. Однако иногда использование операторов switch делает логику

программы более наглядной, в частности, эти операторы идеально подходят для реализации различных *меню*.

Ресторанное меню содержит список блюд, которые может выбирать клиент. Меню в компьютерной программе выполняет сходную задачу: предоставляет пользователю список альтернатив. В листинге 7.4 приведена программа, содержащая информацию о домашних заданиях для студентов. Здесь используется меню, позволяющее учащемуся выбрать интересующие его сведения. (Если хотите увидеть меню этой программы в действии, воспользуйтесь заглушками для определений функций. О заглушках рассказывалось в главе 4.)

Листинг 7.4. Реализация меню с помощью оператора switch

```
// Программа, предоставляющая студентам информацию о домашних заданиях.
#include <iostream>
using namespace std;

void show_assignment();
// Выводит на экран информацию о следующем задании.

void show_grade();
// Запрашивает у студента его номер и выводит его оценку.

void give_hints();
// Выводит подсказки для текущего задания.

int main()
{
    int choice;

    do
    {
        cout << endl
             << "Choose 1 to see the next homework assignment.\n"
             << "Choose 2 for your grade on the last assignment.\n"
             << "Choose 3 for assignment hints.\n"
             << "Choose 4 to exit this program.\n"
             << "Enter your choice and press return: ";
        cin >> choice;

switch (choice)
        {
            case 1:
                show_assignment();
                break;
            case 2:
                show_grade();
                break;
            case 3:
                give_hints();
                break;


            case 4:
                cout << "End of Program.\n";
                break;
            default:
```

```
                cout << "Not a valid choice.\n"
                     << "Choose again.\n";
        }
}while (choice != 4);

    return 0;
}

... Здесь располагаются определения функций show_assignment, show_grade и give_hints. ...
```

Пример диалога

```
Choose 1 to see the next homework assignment.
Choose 2 for your grade on the last assignment.
Choose 3 for assignment hints.
Choose 4 to exit this program.
Enter your choice and press return: 3

Assignment hints:
Analyze the problem.
Write an algorithm in pseudocode.
Translate the pseudocode into a C++ program.

Choose 1 to see the next homework assignment.
Choose 2 for your grade on the last assignment.
Choose 3 for assignment hints.
Choose 4 to exit this program.
Enter your choice and press return: 4
End of Program.
```

Совет программисту: использование в операторах ветвления вызовов функций

Оператор switch и многонаправленный оператор if...else позволяют располагать в каждой ветви по несколько операторов, но в результате весь оператор ветвления становится громоздким и трудно читаемым. Посмотрите на оператор switch, приведенный в листинге 7.4, — в каждой из ветвей для значений 1, 2 и 3 выполняется единственный вызов функции, что делает общую структуру оператора простой и понятной. Если бы код каждой из этих функций размещался прямо в операторе switch, он мог бы разрастись до невероятных размеров, превратившись из оператора в целую программу.

Блоки

Каждая ветвь оператора switch или if...else представляет отдельную подзадачу. Как говорилось ранее, ветви лучше реализовать в виде вызовов функций, и тогда эти подзадачи можно спроектировать, запрограммировать и протестировать по отдельности.

С другой стороны, выполняемые в каждой ветви операции могут быть настолько простыми, что достаточно оформить их как составные операторы. Но и в таком относительно коротком составном операторе может потребоваться собственная

локальная переменная. В качестве примера рассмотрим программу, приведенную в листинге 7.5. Она составляет итоговый счет для заданного количества элементов, приобретенных по указанной цене. При оптовой закупке налог с продажи не взимается — он должен быть уплачен при перепродаже товара розничным покупателям. Поэтому при розничной продаже к сумме, указанной в счете, следует добавить сумму налога. Выбор вида вычислений выполняется оператором `if...else`. Для розничной продажи в вычислениях используется временная переменная `subtotal`, объявленная не на уровне функции или программы, а внутри составного оператора в одной из ветвей оператора `if...else`.

Итак, в программе листинга 7.5, переменная `subtotal` объявлена внутри составного оператора. Такая переменная *локальна* для составного оператора, подобно тому, как переменная, объявленная в теле функции, локальна для этой функции. Вне составного оператора имя `subtotal` при желании можно использовать для какой-нибудь другой цели — оно никак не будет связано с локальной переменной данного оператора. Составной оператор имеет одно преимущество перед определением функции: в нем можно пользоваться всеми переменными, объявленными как вне этого оператора, так и внутри него.

Составной оператор обычно называют *блоком*, а объявленные в нем переменные именуют *локальными переменными этого блока* и говорят, что их областью видимости является блок. (Блоком называют любой код, заключенный в фигурные скобки, независимо от того, имеются ли в нем объявления переменных.)

Блоки

Блоком называется программный код на языке C++, заключенный в фигурные скобки. Объявленные в блоке переменные локальны для этого блока, поэтому их имена могут использоваться вне этого блока для иных целей, например, в качестве имен других переменных.

Листинг 7.5. Блок с локальной переменной

```
// Программа, формирующая счет для оптовой или розничной продажи.
#include <iostream>
using namespace std;
const double TAX_RATE = 0.05;   // 5 % налога


int main()
{
    char sale_type;
    int number;
    double price, total;

    cout << "Enter price $";
    cin >> price;
    cout << "Enter number purchased: ";
    cin >> number;
    cout << "Type W if this is a wholesale purchase.\n"
         << "Type R if this is a retail purchase.\n";
```

```
        << "Then press return.\n";
    cin >> sale_type;

    if ((sale_type == 'W') || (sale_type == 'w'))
    {
        total = price * number;
    }
    else if ((sale_type == 'R') || (sale_type == 'r'))
    {
        double subtotal;   // Локальная переменная для блока.
        subtotal = price * number;
        total = subtotal + subtotal * TAX_RATE;
    }
else
    {
        cout << "Error in input.\n";
    }

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << number << " items at $" << price << endl;
    cout << "Total Bill = $" << total;
    if ((sale_type == 'R') || (sale_type == 'r'))
        cout << " including sales tax.\n";

    return 0;
}
```

Пример диалога

```
Enter price $10.00
Enter number purchased: 2
Type W if this is a wholesale purchase.
Type R if this is a retail purchase.
Then press return.
R
2 items at $10.00
Total Bill = $21.00 including sales tax.
```

Обратите внимание, что тело определения функции представляет собой блок. Блоки, не являющиеся телами функций, не имеют названия, но поскольку мы будем рассматривать их отдельно, то назовем их *блоками операторов*.

Блоки операторов могут быть вложенными друг в друга, при этом к внутренним блокам относятся те же правила локальности переменных. Но применять к ним такие правила не всегда просто, и лучше всего не вкладывать блоки операторов, поскольку это затрудняет чтение программы. Обычно вместо того, чтобы использовать вложенный блок, достаточно просто написать функцию и вызвать ее в том месте, где вы предполагали поместить вложенный блок. В реальных программах блоки операторов применяются очень редко, и в большинстве случаев вызов функции предпочтительнее блока. Но для полноты изложения в следующей врезке приведено правило видимости переменных во вложенных блоках.

Правило видимости переменных во вложенных блоках

Если объявить идентификатор как переменную в двух блоках, вложенных один в другой, получатся две переменные с одним и тем же именем. Одна переменная существует только во внутреннем блоке и вне его является недоступной. Вторая переменная существует только во внешнем блоке и недоступна во внутреннем. Эти переменные различны, и изменения значения одной из них не отражаются на другой.

Ловушка: переменные, случайно оказавшиеся локальными

Если объявить переменную внутри пары фигурных скобок, { }, она станет локальной для заключенного в них блока. Но возможно, вы вовсе не хотите, чтобы переменная была локальной для блока. В этом случае ее следует объявить вне скобок.

Упражнения для самопроверки

19. Что выведет код

```
int first_choice = 1;
switch (first_choice + 1)
{
    case 1:
        cout << "Roast beef\n";
        break;
    case 2:
        cout << "Roast worms\n";
        break;
    case 3:
        cout << "Chocolate ice cream\n";
    case 4:
        cout << "Onion ice cream\n";
        break;
    default:
        cout << "Bon appetit!\n";
}
```

при выполнении в составе полной программы?

20. Что выведет код из предыдущего упражнения, если заменить в нем первую строку следующей:

```
int first_choice = 3;
```

21. А что выведет этот же код, если заменить в нем первую строку следующей:

```
int first_choice = 2;
```

22. Что выведет код, приведенный в упражнении 19, если заменить в нем первую строку такой:

```
int first_choice = 4;
```

23. Что выведет код

```
int number = 22;
{
```

```
    int number = 42;
    cout << number << " ";
}
cout << number;
```

при выполнении в составе полной программы?

24. Хотя мы не советуем программировать таким образом, ниже приведено упражнение с использованием вложенных блоков, которое поможет вам понять, что такое область видимости переменных.

Что выведет код

```
{
    int x = 1;
    cout << x << endl;
    {
        cout << x << endl;
        int x = 2;
        cout << x << endl;
        {
            cout << x << endl;
            int x = 3;
            cout << x << endl;
        }
        cout << x << endl;
    }
    cout << x << endl;
}
```

при выполнении в составе полной программы?

7.3. Циклы в C++

Неверно, что жизнь, это одно проклятие за другим
— это одно и то же проклятие снова и снова.

Эдна Сент-Винсент Миллей

Цикл — это программная конструкция, в которой оператор или последовательность операторов выполняется несколько раз. Примерами циклов являются уже хорошо знакомые вам операторы `while` и `do...while`. Оператор или группа повторяемых в цикле операторов называется *телом* цикла, а каждое повторение тела цикла именуется его *итерацией*. При проектировании любого цикла необходимо решить два главных вопроса — каким должно быть тело цикла и сколько раз оно должно выполняться.

Цикл while

Ниже приведено полное описание синтаксиса оператора `while` и его разновидности — оператора `do...while`. Ключевое различие между этими двумя видами циклов состоит в том, когда именно проверяется логическое выражение, которое определяет условие выполнения цикла. В цикле `while` логическое выражение проверяется до выполнения тела цикла, и если его значением оказывается `false`, то оно вообще

не выполняется. В операторе do...while сначала выполняется тело цикла, а затем проверяется логическое выражение, поэтому тело цикла всегда выполняется как минимум один раз. После первой итерации циклы while и do...while ведут себя одинаково. По завершении каждой итерации логическое выражение проверяется снова, и если оно равно true, выполняется следующая итерация. Как только выражение оказывается равным false, выполнение операторов тела цикла завершается.

Итак, ранее мы сказали, что при выполнении цикла while сначала производится проверка логического выражения, и если его результатом оказывается false, тело цикла никогда не выполняется. У вас могут возникнуть вопросы: зачем вообще нужен цикл, тело которого не выполнено ни разу, и не проще ли всегда применять цикл do...while. Дело в том, что цикл while удобно использовать для суммирования списка чисел, который может оказаться пустым. Например, данный цикл можно применять для подсчета суммы чеков, выписанных в течение месяца, даже если владелец чековой книжки за это время не выпишет ни одного счета. В таком случае тело цикла должно быть выполнено нуль раз.

Ниже показан синтаксис оператора while с телом, состоящим из одного оператора:

```
while (логическое_выражение)
    оператор    // Тело цикла.
```

А синтаксис оператора while с телом, состоящим более чем из одного оператора, выглядит так:

```
while (логическое_выражение)
{
// Тело
    оператор_1
    оператор_2
        .
        .
        .
    оператор_n
}
```

Теперь приведем синтаксис оператора do...while с телом, состоящим из одного оператора:

```
do
    оператор    // Тело цикла.
while (логическое_выражение);
```

и с телом, состоящим более чем из одного оператора:

```
do
{
// Тело цикла.
    оператор_1
    оператор_2
        .
        .
        .
    оператор_n
}while (логическое_выражение);
```

Операторы инкрементирования и декрементирования

Мы уже пользовались оператором инкрементирования для увеличения значения переменной на единицу. Скажем, следующий код:

```
int number = 41;
number++;
cout << number;
```

выводит на экран число 42.

До сих пор мы применяли этот оператор как самостоятельный, не возвращающий значения, но на самом деле он подобен операторам + и - и может использоваться в выражениях. Так, выражение number++ возвращает значение, и его можно применить в таком арифметическом выражении:

```
2*(number++)
```

Выражение number++ сначала возвращает значение переменной number, а затем увеличивает его на единицу. В качестве примера рассмотрим такой код:

```
int number = 2;
int value_produced = 2*(number++);
cout << value_produced << endl;
cout << number << endl;
```

Он выводит на экран два числа:

```
4
3
```

Посмотрите еще раз на выражение 2*(number++). При его вычислении C++ сначала использует значение переменной number, а затем увеличивает ее значение. Таким образом, выражение number++ возвращает значение 2, хотя оператор инкрементирования изменяет значение переменной number (оно становится равным 3). Такое поведение оператора ++ может показаться странным, но иногда оно очень удобно. К тому же, как вы сейчас увидите, оператор ++ может действовать и иначе.

Выражение v++ возвращает значение переменной v и увеличивает его на единицу. Если изменить порядок элементов в выражении, поместив оператор ++ перед переменной, порядок действий тоже изменится. Выражение ++v сначала увеличит значение переменной v на единицу, а затем возвратит это увеличенное значение. Рассмотрим следующий код:

```
int number = 2;
int value_produced = 2*(++number);
cout << value_produced << endl;
cout << number << endl;
```

Он похож на код в предыдущем примере, но оператор ++ располагается перед переменной number. Из-за этого маленького изменения будут выведены совсем другие значения:

```
6
3
```

Обратите внимание, что оба оператора, ++number и number++, выполняют с переменной одно и то же действие: увеличивают ее значение на единицу. Но эти выражения возвращают разные значения. Если оператор ++ стоит перед переменной, он увеличивает ее значение до того, как его вернуть; если же этот оператор располагается за переменной, то инкрементирование выполняется после возврата значения.

В программе, приведенной в листинге 7.6, оператор инкрементирования используется в цикле while для подсчета количества выполнений тела цикла. Подобный контроль итераций является одним из важнейших применений данного оператора.

Листинг 7.6. Оператор инкрементирования в качестве выражения

```
// Программа для подсчета калорий.
#include <iostream>
using namespace std;

int main()
{
    int number_of_items, count,
        calories_for_item, total_calories;

    cout << "How many items did you eat today? ";
    cin >> number_of_items;

    total_calories = 0;
    count = 1;
    cout << "Enter the number of calories in each of the\n"
         << number_of_items << " items eaten:\n";

    while (count++ <= number_of_items)
    {
        cin >> calories_for_item;
        total_calories = total_calories
                       + calories_for_item;
    }

    cout << "Total calories eaten today = "
         << total_calories << endl;
    return 0;
}
```

Пример диалога

```
How many items did you eat today? 7
Enter the number of calories in each of the
7 items eaten:
300 60 1200 600 150 1 120
Total calories eaten today = 2431
```

Все сказанное в этом разделе об операторе инкрементирования относится и к оператору декрементирования, с той разницей, что он не увеличивает значение переменной на единицу, а уменьшает его на это значение. Рассмотрим приведенный ниже код:

```
int number = 8;
int value_produced = number--;
```

```
cout << value_produced << endl;
cout << number << endl;
```

Он выводит два числа:

```
8
7
```

Поменяем местами оператор декрементирования и имя переменной:

```
int number = 8;
int value_produced = --number;
cout << value_produced << endl;
cout << number << endl;
```

Теперь этот код выводит:

```
7
7
```

Выражение `number--` возвращает значение переменной `number` и затем уменьшает его на единицу, а выражение `--number` уменьшает значение этой переменной на единицу и затем возвращает уменьшенное значение.

Учтите, что операторы инкрементирования и декрементирования можно применять только к переменным. Поэтому такие выражения, как `(x + y)++`, `--(x + y)`, `5++` и им подобные, в языке C++ недопустимы.

Упражнения для самопроверки

25. Что выведет код

```
int count = 3;
while (count-- > 0)
    cout << count << " ";
```

при выполнении в составе полной программы?

26. Что выведет код

```
int count = 3;
while (--count > 0)
    cout << count << " ";
```

при выполнении в составе полной программы?

27. Что выведет код

```
int n = 1;
do
    cout << n << " ";
while (n++ <= 3);
```

при выполнении в составе полной программы?

28. Что выведет код

```
int n = 1;
do
    cout << n << " ";
while (++n <= 3);
```

при выполнении в составе полной программы?

Оператор for

Использование операторов while и do...while обеспечивает организацию в программе любых циклов, более того, можно обойтись только оператором while. Однако существует еще один типичный вид циклов, для которого в C++ имеется специальный оператор. При выполнении числовых вычислений очень часто выполняются определенные вычисления для значения 1, затем те же вычисления для значения 2, затем для значения 3 и т. д. до некоторого последнего значения. Например, для того чтобы сложить числа от 1 до 10, нужно чтобы компьютер десять раз выполнил следующий оператор:

```
sum = sum + n;
```

и переменная n в первый раз содержала значение 1, а на каждой следующей итерации ее значение увеличивалось на единицу.

Вот один из способов решения этой задачи:

```
sum = 0;
n = 1;
while (n <= 10)
{
    sum = sum + n;
    n++;
}
```

Хотя здесь отлично справится цикл while, для подобных задач в C++ предусмотрен специальный оператор for (называемый также циклом for). Ниже показано, как с его помощью решается данная задача:

```
sum = 0;
for (n = 1; n <= 10; n++)
    sum = sum + n;
```

Еще ничего не зная о цикле for, вы уже видите: он компактнее, чем цикл while. Рассмотрим его работу подробнее. Прежде всего, обратите внимание, что решения на основе циклов while и for включают одни и те же элементы. Оба они начинаются с оператора присваивания, который устанавливает переменную sum равной 0. В обоих случаях этот оператор выполняется перед циклом. Операторы цикла состоят из следующих элементов:

`n = 1; n <= 10; n++` и `sum = sum + n;`

В цикле for они применяются для тех же целей, что и в цикле while, но первый из них является более компактным, хотя оба цикла выполняет одинаковую работу.

Существуют и другие возможности, но в этой книге мы будем пользоваться только операторами for, управляемыми одной переменной, — в нашем примере это переменная n. Давайте рассмотрим правила написания циклов for.

Оператор for начинается с ключевого слова for, за которым в скобках следуют три элемента, указывающих компилятору, что делать с управляющей переменой цикла. Начало цикла for выглядит так:

for (*инициализирующее_выражение*; *логическое_выражение*; *модифицирующее_выражение*)

Первое выражение указывает на то, как инициализируется переменная цикла, второе используется для проверки условия окончания цикла, а третье указывает,

как переменная цикла обновляется после каждой его итерации. В нашем примере цикл for начинается следующим образом:

```
for (n = 1; n <= 10; n++)
```

Выражение n = 1 указывает, что переменная n должна быть инициализирована значением 1. Выражение n <= 10 говорит о том, что цикл должен продолжаться, пока значение переменной n меньше или равно 10. А последнее выражение, n++, используется для указания на то, что после каждого выполнения тела цикла значение переменной n должно увеличиваться на единицу.

Все эти три выражения в начале цикла for должны быть разделены двумя (и только двумя!) точками с запятой, так что не поддавайтесь искушению поставить точку с запятой после третьего выражения. (Поскольку это не операторы, а именно выражения, их не нужно завершать точкой с запятой, которая используется только в качестве разделителя.)

На рис. 7.4 приведен синтаксис оператора for и показано его действие на примере совместно с альтернативным циклом while. Обратите внимание, что в операторе for, как и в соответствующем операторе while, условие прекращения цикла проверяется до первой итерации. Поэтому и тот, и другой цикл допускает, что его тело будет выполнено нуль раз.

В листинге 7.7 показан пример цикла for в составе полной (хотя и очень простой) программы. Приведенный цикл похож на тот, что мы только что рассмотрели, но в нем имеется один новый элемент. Здесь переменная n объявляется при инициализации, поэтому ее объявление располагается внутри цикла for. Одной из дополнительных возможностей данного цикла является совмещенное объявление и инициализация переменой цикла. Это очень удобно в тех случаях, когда переменная используется только внутри цикла, но если она должна быть доступна и вне его, ее следует объявить за пределами цикла.

Листинг 7.7. Пример использования оператора for

```
// Демонстрирует цикл for.
#include <iostream>
using namespace std;

int main()
{
    int sum = 0;
for (int n = 1; n <= 10; n++)  // Заметьте, что переменная n является
                               // локальной переменной тела цикла.
        sum = sum + n;

    cout << "The sum of the numbers 1 to 10 is "
         << sum << endl;
    return 0;
}
```

Пример диалога

```
The sum of the numbers 1 to 10 is 55
```

Оператор for

Синтаксис

```
for (инициализирующее_выражение; логическое_выражение; модифицирующее_выражение)
    оператор_тела
```

Пример

```
for (number = 100; number >= 0; number--)
    cout << number
         << " bottles of beer on the shelf.\n";
```

Альтернативный цикл while

Синтаксис

```
инициализирующее_выражение;
while (логическое_выражение)
{
    оператор_тела
    модифицирующее_выражение;
}
```

Пример

```
number = 100;
while (number >= 0)
{
    cout << number
         << " bottles of beer on the shelf.\n";
    number--;
}
```

Вывод

```
100 bottles of beer on the shelf.
99 bottles of beer on the shelf.
                .
                .
                .
0 bottles of beer on the shelf.
```

Рис. 7.4. Оператор for

Стандарт ANSI C++ требует, чтобы объявление в инициализационном выражении цикла for считалось локальным для этого цикла. Ранние компиляторы C++ не следовали этому правилу, поэтому выясните, как ваш компилятор интерпретирует объявленные таким образом переменные. А для того чтобы программы получались максимально переносимыми, лучше не полагаться на то или иное поведение компилятора в этом вопросе, а строго выполнять требования стандарта.

Кроме того, в соответствии со стандартом ANSI объявление в инициализационном выражении цикла for должно считаться локальным для блока, составляющего тело этого цикла. Скорее всего, следующее поколение компиляторов, будет соответствовать данному требованию, но компиляторы, доступные в настоящее время, могут ему не удовлетворять.

Нашему описанию цикла for немного не хватает универсальности. Три выражения в начале оператора for могут быть любыми выражениями C++ и включать более (или даже менее!) одной переменной. Однако в примерах нашей книги они всегда будут содержать только одну переменную.

В операторе for, приведенном в листинге 7.7, тело составляет единственный оператор присваивания:

```
sum = sum + n;
```

В общем случае телом цикла может быть любой оператор, в том числе и составной. Это позволяет поместить в тело цикла for несколько операторов. В таком случае синтаксис данного цикла будет следующим:

```
for (инициализирующее_выражение; логическое_выражение; модифицирующее_выражение)
{
    оператор_1
    оператор_2
        .
        .
        .
    оператор_n
}
```

Например:

```
for (int number = 100; number >= 0; number--)
{
    cout << number
         << " bottles of beer on the shelf.\n";
    if (number > 0)
        cout << "Take one down and pass it around.\n";
}
```

В приведенных до сих пор циклах for после каждой итерации значение управляющей переменной увеличивалось или уменьшалось на единицу. Однако возможны и другие способы ее обновления. Она может увеличиваться на два или на три либо на дробное значение, если это переменная типа double. В частности, допустимы такие операторы for:

```
for (n = 1; n <= 10; n = n + 2)
    cout << "n is now equal to " << n << endl;

for (n = 0; n > -100; n = n - 7)
    cout << "n is now equal to " << n << endl;

for (double size = 0.75; size <= 5; size = size + 0.05)
    cout << "size is now equal to " << size << endl;
```

Более того, обновление вообще не обязательно должно быть сложением или вычитанием, а инициализация не всегда означает установку переменной равной целому числу. Инициализировать и изменять переменную можно любыми способами. Вот еще один пример возможного цикла for:

```
for (double x = pow(y, 3.0); x > 2.0; x = sqrt(x))
    cout << "x is now equal to " << x << endl;
```

Ловушка: лишняя точка с запятой в операторе for

Не ставьте точку с запятой после закрывающей круглой скобки в цикле for. Следующий пример показывает, что произойдет в этом случае.

```
for (int count = 1; count <= 10; count++);   // Лишняя точка с запятой.
    cout << "Hello\n";
```

Если не заметить лишнюю точку с запятой, можно решить, что этот цикл десять раз выведет на экран слово Hello. Если же вы ее заметите, то возможно, предположите, что компилятор выдаст сообщение об ошибке. Ничего подобного! При компиляции программы с таким циклом, вы не получите никаких сообщений от компилятора. А после ее выполнения на экране будет отображено одно слово Hello вместо десяти. Что же произошло? Ответ на этот вопрос очень прост, но для того чтобы его получить, вернемся немного назад.

Один из способов создания оператора в C++ заключается в том, чтобы поместить после выражения точку с запятой. Например, если поставить точку с запятой после выражения x++, оно превратится в оператор

```
x++;
```

При введении точки с запятой после «ничего» тоже получится оператор. Иными словами, одиночная точка с запятой сама по себе является оператором, называемым *пустым оператором*. Он не выполняет никакого действия, но все же это оператор. Поэтому следующая строка:

```
for (int count = 1; count <= 10; count++);
```

представляет собой полный и синтаксически правильный цикл for, тело которого — пустой оператор.

Приведенный цикл выполнится десять раз, но при этом ровным счетом ничего не произойдет, поскольку его тело составляет пустой оператор. Данный цикл ничего не делает, а точнее, делает «ничего» десять раз!

Теперь вернемся к нашему примеру и посмотрим на строку с комментарием Лишняя точка с запятой. Из-за наличия лишней точки с запятой цикл for оканчивается этой же строкой и имеет пустое тело, а потому ничего не делает. После его завершения выполняется следующий оператор cout, выводящий на экран слово Hello один раз:

```
cout << "Hello\n";
```

Позже вы узнаете, для чего может быть полезен цикл for с пустым телом, но пока это просто ошибка, допущенная по невнимательности.

Каким циклом пользоваться

При проектировании циклов выбор подходящего оператора C++ обычно лучше отложить на конец этой процедуры. Сначала разработайте цикл с использованием псевдокода и только после этого переведите его на язык C++. В данный момент

будет легче всего решить, какой из операторов цикла лучше всего подходит для данной задачи.

Если цикл содержит числовые вычисления с использованием переменной, равномерно изменяющейся на каждой итерации, используйте цикл for. Как правило, он удобен для всех задач, в которых выполняются числовые вычисления.

В большинстве остальных случаев следует применять цикл while или do...while. Выбрать один из них очень просто: если тело цикла должно выполниться как минимум один раз, воспользуйтесь циклом do...while, а если возможны ситуации, когда тело цикла не должно выполниться ни разу, применяйте цикл while. Типичной ситуацией, в которой больше подходит цикл while, является чтение ввода, когда данных может вообще не оказаться. Например, если программа считывает список экзаменационных оценок, некоторые студенты могут не сдать ни одного экзамена, и тогда программа столкнется с пустым списком. В этом случае следует использовать цикл while.

Упражнения для самопроверки

29. Что выведет код

```
for (int count = 1; count < 5; count++)
    cout << (2 * count) << " ";
```

при выполнении в составе полной программы?

30. Что выведет код

```
for (int n = 10; n > 0; n = n - 2)
{
    cout << "Hello";
    cout << n << endl;
}
```

при выполнении в составе полной программы?

31. Что выведет код

```
for (double sample = 2; sample > 0; sample = sample - 0.5)
    cout << sample << " ";
```

при выполнении в составе полной программы?

32. Какой цикл больше подходит в каждой из следующих ситуаций:

    а) суммирование последовательности, например 1/2 + 1/3 + 1/4 + 1/5 + ... + +1/10;

    б) чтение списка экзаменационных оценок студента;

    в) чтение количества дней, за которые сотрудники фирмы получили больничный лист;

    г) тестирование функции для проверки ее поведения с разными значениями аргументов.

33. Перепишите следующие циклы как циклы for.

    а)
```
int i = 1;
while (i <= 10)
```

```
    {
        if (i < 5 && i != 2)
        cout << 'X';
        i++;
    }
```

б)
```
int i = 1;
while (i <=10)
{
    cout << 'X';
    i = i + 3;
}
```

в)
```
long m = 100;
do
{
    cout << 'X';
    m = m + 100;
} while (m < 1000);
```

34. Что выведет следующий цикл:

```
int n = 1024;
int log = 0;
for (int i = 1; i < n; i = i * 2)
    log++;
cout << n << " " << log << endl;
```

Покажите связь между значениями переменных n и log.

35. Что выведет следующий цикл:

```
int n = 1024;
int log = 0;
for (int i = 1; i < n; i = i * 2);
    log++;
cout << n << " " << log << endl;
```

Прокомментируйте его.

36. Что выведет следующий цикл:

```
int n = 1024;
int log = 0;
for (int i = 0; i < n; i = i * 2)
    log++;
cout << n << " " << log << endl;
```

Прокомментируйте его.

Ловушка: неинициализированные переменные и бесконечные циклы

В главе 2, где впервые рассматривались циклы while и do...while, мы предупреждали о двух потенциальных проблемах, связанных с циклами. Во-первых, при составлении цикла следует убедиться, что все использующиеся в нем переменные инициализируются (то есть получают значения) до начала его выполнения. Это кажется очевидным, однако на практике такой момент легко упустить. Во-вторых,

необходимо следить за тем, чтобы цикл не оказался бесконечным. Оба эти предостережения относятся и к циклу for.

Оператор break

Вы уже пользовались оператором break для выхода из оператора switch. Точно так же он применяется и для выхода из цикла. Иногда логика программы требует выхода из цикла раньше его нормального завершения. Например, цикл может содержать проверку введенных данных, и если они окажутся неправильными, возникнет необходимость в немедленном выходе из цикла. В листинге 7.8 приведен код, считывающий список отрицательных чисел и вычисляющий их сумму, помещаемую в переменную sum. Если пользователь вводит десять отрицательных чисел, цикл завершается нормально, если же пользователь забудет набрать знак минус, цикл немедленно завершится с помощью оператора break.

Оператор break

Оператор break может использоваться для выхода из оператора цикла. Когда он выполнятся, цикл немедленно завершается и выполнение программы продолжается с оператора, следующего за оператором цикла. Оператор break можно применять в любом из циклов: while, do...while или for. Это тот же самый оператор, которым мы пользовались для выхода из оператора switch.

Листинг 7.8. Пример использования оператора break в цикле

```
// Складывает 10 отрицательных чисел.
#include <iostream>
using namespace std;

int main()
{
    int number, sum = 0, count = 0;
    cout << "Enter 10 negative numbers:\n";

    while (++count <= 10)
    {
        cin >> number;

        if (number >= 0)
        {
            cout << "ERROR: positive number"
                 << " or zero was entered as the\n"
                 << count << "th number! Input ends "
                 << "with the " << count << "th number.\n"
                 << count << "th number was not added in.\n";
            break;
        }

        sum = sum + number;
    }
```

продолжение

Листинг 7.8 (*продолжение*)

```
    cout << sum << " is the sum of the first "
         << (count - 1) << " numbers.\n";

    return 0;
}
```

Пример диалога

```
Enter 10 negative numbers:
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10
ERROR: positive number or zero was entered as the
4th number! Input ends with the 4th number.
4th number was not added in.
-6 is the sum of the first 3 numbers.
```

Ловушка: оператор break во вложенных циклах

Оператор break завершает только самый внутренний из содержащих его циклов. Если в программе выполняется цикл в цикле и внутренний цикл содержит оператор break, этот оператор завершит только внутренний цикл, а внешний будет продолжаться.

Упражнения для самопроверки

37. Что выведет код

```
int n = 5;
while (--n > 0)
{
    if (n == 2)
    break;
    cout << n << " ";
}
cout << "End of Loop.";
```

при выполнении в составе полной программы?

38. Что выведет код

```
int n = 5;
while (--n > 0)
{
    if (n == 2)
        exit(0);
    cout << n << " ";
}
cout << "End of Loop.";
```

при выполнении в составе полной программы?

39. Что делает оператор break? Где его можно размещать?

7.4. Разработка циклов

Она ходит и ходит по кругу, и никто не знает, где она остановится.

Традиционные слова карнавального зазывалы

В ходе разработки цикла определяются три элемента:

- тело цикла;
- инициализирующие выражения;
- условия завершения цикла.

В этом разделе мы рассмотрим два распространенных типа задач, решаемых с помощью цикла, и выясним, как для каждого из них разрабатываются три указанных элемента.

Циклы для сумм и произведений

Многие типичные задачи требуют чтения списка чисел и вычисления их суммы. Если количество чисел известно заранее, эта задача решается с помощью такого псевдокода:

```
sum = 0;
повторить следующие вычисления this_many раз:
    cin >> next;
    sum = sum + next;
конец цикла.
```

Значением переменной this_many является количество складываемых чисел. Сумма накапливается в переменной sum.

Приведенный псевдокод проще всего реализовать в виде цикла for:

```
int sum = 0;
for (int count = 1; count <= this_many; count++)
{
    cin >> next;
    sum = sum + next;
}
```

Обратите внимание, что к моменту выполнения следующего оператора тела цикла переменная sum уже должна содержать значение:

```
sum = sum + next;
```

Поэтому ее следует инициализировать до начала цикла. Чтобы понять, какое значение ей нужно присвоить, подумайте о действиях, происходящих после первой итерации цикла. В результате прибавления первого числа значением переменной sum должно стать это число. Иными словами, после первого выполнения тела цикла значение sum + next должно быть равно next. Для этого переменную sum нужно инициализировать значением 0.

Тем же способом может формироваться и произведение чисел. Вот пример:

```
int product = 1;
for (int count = 1; count <= this_many; count++)
```

```
{
    cin >> next;
    product = product * next;
}
```

Переменной product должно быть присвоено начальное значение, но на этот раз — не нуль. Если инициализировать переменную product значением 0, после первой итерации цикла она по-прежнему будет равна нулю (как и после всех последующих итераций). Поэтому правильно присвоить ей значение 1. После первой итерации цикла переменная product должна быть равна первому прочитанному числу (хранящемуся в переменной next), а это возможно только в том случае, если значение переменной next будет умножено на единицу.

Повторение заданное количество раз

Оператор for может использоваться для создания цикла, тело которого повторяется заранее известное количество раз.

Псевдокод

```
Повторить следующее столько_раз
    тело_цикла
```

Эквивалент оператора for

```
for (int счетчик = 1; счетчик <= столько_раз; счетчик ++)
    тело_цикла
```

Пример

```
for (int count = 1; count <= 3; count++)
    cout << "Hip, Hip, Hurray\n";
```

Завершение циклов

Существует четыре распространенных способа завершения цикла, в котором выполняется ввод данных. Мы расскажем о них по порядку.

1. В начале списка входных значений указывается его размер.
2. Перед каждой итерацией запрашивается ответ пользователя.
3. Список входных значений завершается сигнальным значением.
4. Заканчиваются входные данные.

Если программа может заранее определить размер списка входных значений, запросив его у пользователя или каким-нибудь другим способом, для чтения в точности *n* значений можно использовать цикл «повторить *n* раз». Это первый из перечисленных выше методов.

Второй метод заключается в том, чтобы после каждой итерации просто запрашивать у пользователя, следует ли повторить выполняемую операцию. Например:

```
sum = 0;
cout << "Are there any numbers in the list? (Type\n"
     << "Y and Return for Yes, N and Return for No): ";
char ans;
```

```
cin >> ans;
while ((ans == 'Y') || (ans == 'y'))
{
    cout << "Enter number: ";
    cin >> number;
    sum = sum + number;
    cout << "Are there any more numbers? (Type\n"
         << "Y for Yes, N for No. End with Return.): ";
    cin >> ans;
}
```

Однако при вводе длинного списка применение такого метода утомительно для пользователя. Поэтому гораздо лучше использовать специальный сигнал останова.

Пожалуй, лучше всего завершать цикл, считывающий список вводимых с клавиатуры значений, некоторым *сигнальным значением*. Это особое значение, отличное от всех возможных значений вводимых данных. Так, если цикл считывает список положительных чисел, отрицательное число может использоваться как сигнальное значение, указывающее на конец списка. Вот вариант такого решения:

```
cout << "Enter a list of nonnegative integers.\n"
     << "Place a negative integer after the list.\n";
sum = 0;
cin >> number;
while (number >= 0)
{
    sum = sum + number;
    cin >> number;
}
```

Обратите внимание, что последнее число в списке считывается, но не прибавляется к сумме. Для сложения чисел 1, 2 и 3 пользователь добавляет в конец списка любое отрицательное число:

```
1 2 3 -1
```

Последнее значение, то есть -1, считывается, но не прибавляется к значению переменной sum.

Для подобного использования сигнального значения нужна уверенность, что как минимум одно значение типа данных не может присутствовать в списке входных значений, а значит, может играть роль сигнального. Но когда список состоит из целочисленных значений и может содержать любые числа, то не существует значения, которое можно было бы использовать в качестве сигнального, и для завершения цикла следует выбрать иной метод.

Если данные считываются из файла, можно использовать сигнальное значение, но проще проверять, достигнут ли конец файла, и завершать цикл, когда все входные данные прочитаны. Этот метод описан в главе 5.

Все приведенные методы завершения цикла являются частными случаями следующих более универсальных технологий:

- циклы, управляемые счетчиком;
- запрос перед итерацией;
- выход по флагу.

Цикл, управляемый счетчиком, — это цикл, для которого количество итераций известно до его начала. В качестве примера можно назвать рассмотренный выше цикл ввода данных, обрабатывающий список, в начале которого указан его размер. Все циклы типа «повторить заданное количество раз» являются циклами, управляемыми счетчиком.

Вторую технологию, *запрос перед итерацией*, мы тоже уже рассмотрели. Чаще всего она используется для циклов, реализующих ввод данных с клавиатуры.

Еще мы описали циклы, оканчивающиеся после ввода специального сигнального значения. В этом разделе рассматривался пример, когда программа осуществляет ввод неотрицательных чисел и заканчивает его при вводе пользователем отрицательного числа. Это частный случай более универсальной технологии, называемой *выходом по флагу*. Переменная, изменение значения которой указывает на некоторое событие, часто называется *флагом*. В нашем примере цикла ввода флагом была переменная `number`. Ей присваивалось введенное пользователем число, и когда ее значение становилась отрицательным, это означало, что цикл должен завершиться.

Окончание цикла ввода данных из файла по достижении конца файла является еще одним примером технологии «выхода по флагу». В этом случае флаг выхода определяется системой, которая отслеживает, достиг ли процесс чтения конца файла.

Флаг может использоваться и для прекращения выполнения циклов, не реализующих ввод данных. В частности, следующий цикл ищет среди студентов возможного консультанта. Список студентов пронумерован, начиная с 1. Цикл просматривает данные об успеваемости и останавливается, найдя студента, который получил высокую оценку. Высокой считается оценка 90 или более баллов. Предполагается, что функция `compute_grade`, определяющая оценку очередного студента, уже определена.

```
int n = 1;
grade = compute_grade(n);
while (grade < 90)
{
    n++;
    grade = compute_grade(n);
}
cout << "Student number " << n << " may be a tutor.\n"
     << "This student has a score of " << grade << endl;
```

В этом примере флагом служит переменная `grade`.

Здесь продемонстрирована типичная проблема, связанная с разработкой циклов. Что произойдет, если ни один студент не получит оценку 90 или более баллов? Ответ зависит от определения функции `compute_grade`. Если она принимает любые числа, цикл может оказаться бесконечным. Хуже того, если функция возвращает, скажем, `100` для всех аргументов, которые не являются номерами студентов, программа может назначить инструктором несуществующего студента. Если есть вероятность, что при определенных входных данных цикл может оказаться бесконечным или выполняться большее количество раз, чем того требует постановка

задачи, в него нужно включить дополнительную проверку, которая будет гарантировать своевременное прекращение цикла. Например, для приведенного выше цикла более удачным будет следующее условие (переменной number_of_students присвоено количество студентов в классе):

```
int n = 1;
grade = compute_grade(n);
while ((grade < 90) && (n < number_of_students))
{
    n++;
    grade = compute_grade(n);
}
if (grade >= 90)
    cout << "Student number " << n << " may be a tutor.\n"
        << "This student has a score of " << grade << endl;
else
    cout << "No student has a high score.";
```

Вложенные циклы

Программа, приведенная в листинге 7.9, помогает отслеживать воспроизводство одного из исчезающих подвидов грифов. Ежегодно в районе их обитания специалисты по охране окружающей среды подчитывают количество яиц в гнездах. Программа принимает собранные данные и выводит общее число яиц, содержащихся во всех обследованных гнездах.

Листинг 7.9. Вложенные циклы

```
// Определяет общее количество яиц в гнездах грифов по отчетам
// всех наблюдателей в районе обследования.
#include <iostream>
using namespace std;

void instructions();

void get_one_total(int& total);
// Предусловие: пользователь ввел список, включающий количество
// яиц в каждом гнезде и завершающийся отрицательным числом.
// Постусловие: total – суммарное число яиц в гнездах.

int main()
{
    instructions();

    int number_of_reports;
    cout << "How many conservationist reports are there? ";
    cin >> number_of_reports;

    int grand_total = 0, subtotal, count;
    for (count = 1; count <= number_of_reports; count++)
    {
        cout << endl << "Enter the report of "
             << "conservationist number " << count << endl;
get_one_total(subtotal);
```

продолжение

Листинг 7.9 (*продолжение*)

```
        cout << "Total egg count for conservationist "
             << " number " << count << " is "
             << subtotal << endl;
        grand_total = grand_total + subtotal;
    }

    cout << endl << "Total egg count for all reports = "
         << grand_total << endl;

    return 0;
}

// Используем библиотеку классов iostream.
void instructions()
{
    cout << "This program tallies conservationist reports\n"
         << "on the green-necked vulture.\n"
         << "Each conservationist's report consists of\n"
         << "a list of numbers. Each number is the count of\n"
         << "the eggs observed in one"
         << " green-necked vulture nest.\n"
         << "This program then tallies"
         << " the total number of eggs.\n";
}

// Используем библиотеку классов iostream.
void get_one_total(int& total)
{
    cout << "Enter the number of eggs in each nest.\n"
         << "Place a negative integer"
         << " at the end of your list.\n";

    total = 0;
    int next;
    cin >> next;
    while (next >= 0)
    {
        total = total + next;
        cin >> next;
    }
}
```

Пример диалога

```
This program tallies conservationist reports
on the green-necked vulture.
Each conservationist's report consists of
a list of numbers. Each number is the count of
the eggs observed in one green-necked vulture nest.
This program then tallies the total number of eggs.
How many conservationist reports are there? 3

Enter the report of conservationist number 1
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
```

```
1 0 0 2 -1
Total egg count for conservationist  number 1 is 3

Enter the report of conservationist number 2
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
0 3 1 -1
Total egg count for conservationist  number 2 is 4

Enter the report of conservationist number 3
Enter the number of eggs in each nest.
Place a negative integer at the end of your list.
-1
Total egg count for conservationist  number 3 is 0

Total egg count for all reports = 7
```

Отчет каждого наблюдателя состоит из списка чисел, представляющих количество яиц в обследованных им гнездах. Функция get_one_total типа void считывает отчет одного наблюдателя и вычисляет общее количество обнаруженных им яиц. В конце списка вводится отрицательное число, указывающее на окончание списка, оно служит сигнальным значением. Функция get_one_total включена в цикл for и вызывается по одному разу для каждого отчета.

Тело цикла может включать любые операторы, а также вложенные циклы. Программа в листинге 7.9 содержит цикл в цикле. Обычно подобный код не считается содержащим вложенный цикл, поскольку внутренний цикл находится в функции, вызываемой из внешнего цикла. Он наглядно показывает, что вложенные циклы ничем не отличаются от любых других. В листинге 7.10 приведена еще одна версия программы, на этот раз с явно вложенными циклами. Внешний цикл здесь выполняется по одному разу для каждого значения переменой count от 1 до number_of_reports. И на каждой его итерации полностью выполняется внутренний цикл while.

Функционально две версии программы для подсчета количества яиц грифов эквивалентны. Обе они осуществляют один и тот же диалог с пользователем. Однако большинство людей сочтет версию из листинга 7.9 более понятной, поскольку тело внешнего цикла содержит вызов функции. Этот вызов представляет вычисление промежуточного итога как одну операцию, а не как цикл.

Листинг 7.10. Явно вложенные циклы

```
// Определяет общее количество яиц в гнездах грифов по отчетам
// всех наблюдателей в районе обследования.
#include <iostream>
using namespace std;

void instructions();

int main()
{
    instructions();

    int number_of_reports;
```

продолжение

Листинг 7.10 (*продолжение*)

```
    cout << "How many conservationist reports are there? ";
    cin >> number_of_reports;

    int grand_total = 0, subtotal, count;
    for (count = 1; count <= number_of_reports; count++)
    {
        cout << endl << "Enter the report of "
             << "conservationist number " << count << endl;
        cout << "Enter the number of eggs in each nest.\n"
             << "Place a negative integer"
             << " at the end of your list.\n";
        subtotal = 0;
        int next;
        cin >> next;
        while (next >= 0)
        {
            subtotal = subtotal + next;
            cin >> next;
        }
        cout << "Total egg count for conservationist "
             << " number " << count << " is "
             << subtotal << endl;
        grand_total = grand_total + subtotal;
    }

    cout << endl << "Total egg count for all reports = "
         << grand_total << endl;
    return 0;
}
```

... Определение функции instructions такое же, как в листинге 7.9. ...

Превращайте тело цикла в вызов функции

Если тело цикла содержит сложные вычисления или вложенный цикл, превращайте его в вызов функции. Тем самым вы отделите структуру тела цикла от структуры остальной части программы, а программируемую задачу разделите на две меньшие подзадачи.

Упражнения для самопроверки

40. Напишите цикл, десять раз выводящий на экран слово `Hello`.
41. Напишите цикл, считывающий список четных чисел и вычисляющий их общую сумму. Список завершается сигнальным значением. Вам нужно решить, какое значение подойдет для этой цели.
42. Что выведут следующие вложенные циклы:

```
int n, m;
for (n = 1; n <= 10; n++)
    for (m = 10; m >= 1; m--)
    cout << n << " * " << m
         << " = " << n*m << endl;
```

Отладка циклов

Как бы тщательно не была разработана программа, все равно она может содержать ошибки. Существует ряд типичных ошибок, характерных именно для циклов. Большая их часть связана с первой или последней итерацией цикла. Если его тело выполняется на один раз больше или меньше, чем следует, возникает ошибка, называемая *ошибкой на единицу*; она является одной из самых распространенных. Не путайте операторы < и <=, контролируйте правильную инициализацию цикла, помните, что иногда тело цикла не должно выполниться ни разу, и проверяйте, верно ли обрабатывается такая возможность.

Бесконечные циклы обычно получаются из-за ошибки в логическом выражении, определяющем условие окончания цикла. Следите, чтобы выполнялось нужное сравнение, и знаки > и < случайно не поменялись местами. Часто бесконечные циклы возникают из-за проверки на равенство там, где требовалась сравнение «больше» или «меньше». Для значений типа `double` проверка на равенство не всегда дает корректный ответ, поскольку сравниваются приближенные значения величин. Но даже для значений типа `int` это рискованная проверка, так как она удовлетворяется в единственном случае.

Если цикл проверен и перепроверен, а ошибка все равно не найдена и программа ведет себя неправильно, необходимо более сложное тестирование. Прежде всего, убедитесь, что ошибка находится именно в цикле. Ведь то, что программа ведет себя неверно, еще не означает, что ошибка происходит там, где вы думаете. Если программа разделена на две функции, то примерное место возникновения ошибки (или ошибок) обычно определить не сложно.

Выяснив, что ошибка действительно в цикле, проследите за тем, каким образом он изменяет значения переменных. Это позволит определить действия цикла и понять, что в нем неверно. Наблюдение за изменением значения переменной во время выполнения цикла называется *трассировкой* этой переменной. Во многих системах имеются отладочные утилиты, позволяющие трассировать переменные, не внося изменений в программу. Если в вашей системе есть такая утилита, стоит научиться ею пользоваться. Если же она отсутствует, можно трассировать переменные с помощью временных объектов `cout` в теле цикла (не забудьте удалить их по окончании тестирования!). С их помощью значение переменой будет выводиться на экран на каждом шаге цикла.

В качестве примера рассмотрим следующий нуждающийся в отладке фрагмент кода:

```
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
}
// В переменной product содержится
// произведение чисел от 2 до 5.
```

Комментарий в конце цикла поясняет, что должен делать этот цикл, Однако мы проверили его и знаем, что переменная product получает неверное значение. Теперь необходимо понять, что же именно происходит не так. Для этого будем трассировать переменные next и product. Если у вас имеется отладочная утилита, воспользуйтесь ею. Если нет — трассировать эти переменные можно, вставив в цикл следующий оператор cout:

```
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
cout << "next = " << next
        << " product = " << product << endl;
}
```

Трассируя переменные next и product, мы обнаружим, что после первой итерации значения обеих переменных равны 3. Поэтому ясно, что мы перемножили числа от 3 до 5 и пропустили умножение на 2.

Эту ошибку можно исправить как минимум двумя способами. Проще всего инициализировать переменную next значением 1, а не 2. Тогда после первого приращения в цикле она получит значение 2, с которого, как и положено, начнется умножение. В качестве альтернативы можно поместить оператор инкрементирования после умножения, вот так:

```
int next = 2, product = 1;
while (next < 5)
{
product = product * next;
    next++;
}
```

Предположим, что мы исправили ошибку, переместив оператор next++, как показано выше. Однако работа еще не закончена. Далее следует проверить модифицированный код, чтобы убедиться, что теперь он работает правильно. После повторной трассировки переменных выясняется, что процесс начинается правильно, но заканчивается после умножения на 4, а умножение на 5 вообще не выполняется. Это означает, что в логическом выражении нам следует использовать оператор <=, а не <. Правильный код будет таким:

```
int next = 2, product = 1;
while (next <= 5)
{
    product = product * next;
    next++;
}
```

После каждого изменения программы ее нужно снова протестировать. Никогда не полагайтесь на правильность внесенных изменений. То что найден один нуждающийся в исправлении фрагмент кода, еще не означает, что отсутствуют другие. Кроме того, как показывает данный пример, исправление одной части программы может потребовать изменения и других ее частей.

Тестирование цикла

Каждый цикл следует протестировать с входными значениями для каждого из следующих вариантов поведения цикла: нуль итераций тела цикла; одна итерация; максимальное количество итераций; максимальное количество итераций минус одна. (Это лишь минимальный набор тестируемых ситуаций. Кроме того, следует провести другие тесты, специфические для конкретного цикла.)

Описанные простейшие технологии отладки помогут найти ошибки, которые могут вкрасться даже в скрупулезно разработанную программу. Однако никакая самая тщательная отладка не превратит плохо спроектированную и написанную программу в надежную и читабельную. Если программа или алгоритм непонятны или плохо работают, не пытайтесь их исправить. Лучше начните все сначала с чистого листа. Созданная в результате программа будет значительно более читабельной и менее вероятно, что в ней окажутся скрытые ошибки. И что еще несомненно, хотя и менее очевидно, особенно для новичка, – это тот факт, что создать программу заново быстрее, чем доводить до ума плохо спроектированную и написанную программу. Потраченное время не потеряно – усвоенные уроки помогут разработать лучшую программу быстрее, чем если бы вы только приступали к работе над данной задачей.

Отладка очень плохой программы

Если программа спроектирована и написана очень плохо, не пытайтесь ее отлаживать, лучше откажитесь от нее и начните все сначала.

Упражнения для самопроверки

43. Что означает трассировать переменную? Как это делается?
44. Что такое ошибка на единицу?
45. Нам нужно поставить изгородь длиной 100 футов. Колья изгороди должны располагаться через каждые 10 футов. Сколько их потребуется? Почему эта задача в книге по программированию не столь неуместна, как кажется? С какой типичной для программирования проблемой она связана?

Резюме

- Логические выражения вычисляются подобно арифметическим.
- Большинство современных компиляторов поддерживает тип данных `bool`, используемый со значениями `true` и `false`.
- Можно написать функцию, возвращающую значение `true` или `false`. Ее вызов будет использоваться в логическом выражении в операторе `if...else` или в любом другом месте, где допускается использование логических выражений.

- Одним из подходов к решению задачи или подзадачи является запись условий и действий, выполняемых для каждого из условий. Такое решение в C++ можно реализовать с помощью многонаправленного оператора `if...else`.
- Оператор `switch` удобен при реализации меню для пользователей программы.
- В операторах, выполняющих многонаправленное ветвление, таких как `switch` и `if...else`, желательно использовать вызовы функций.
- Блоком называется составной оператор, который может содержать объявления переменных. Переменные, объявленные в нем, локальны для этого блока. Помимо прочего блоки могут использоваться в операторах ветвления (двунаправленного или многонаправленного), например, в операторе `if...else`.
- Цикл `for` может использоваться для получения эквивалента инструкции «повторить тело цикла *n* раз».
- Существует четыре распространенных метода завершения цикла, выполняющего ввод массива данных: в начале списка указывается его размер; перед каждой итерацией задается вопрос; список завершается сигнальным значением; заканчиваются входные данные.
- Обычно лучше всего разрабатывать циклы в псевдокоде, отложив выбор конкретного механизма реализации цикла на C++. Когда алгоритм готов, выбор сделать намного проще.
- Один из способов сделать вложенные циклы более читабельными заключается в реализации тела цикла в виде вызова функции.
- Всегда проверяйте циклы на предмет правильной инициализации используемых в них переменных.
- Обязательно проверяйте, не выполняется ли тело цикла на один раз больше или меньше, чем требуется.
- При отладке циклов помогает трассировка используемых в них ключевых переменных.
- Если программа или алгоритм слишком непонятны или очень плохо работают, не пытайтесь их исправлять. Лучше начните все сначала.

Ответы к упражнениям для самопроверки

1. а) `true`

   б) `true`. Заметьте, что выражения а), б) означают одно и то же. Поскольку операторы `==` и `<` имеют более высокий приоритет, чем оператор `&&`, скобки в выражении не обязательны, однако они облегчают чтение. Большинство людей сочтет выражение а) более читабельным, чем выражение б), хотя означают они одно и то же.

   в) `true`

   г) `true`

д) false. Поскольку значением первого подвыражения (count == 1) является false, мы знаем, что все выражение равно false, и для этого нам не нужно определять значение второго подвыражения. Таким образом, неважно, каковы значения переменных x и y. В этом и заключается выполняемое C++ сокращенное вычисление выражения.

е) true. Поскольку значением первого подвыражения (count < 10) является true, мы знаем, что все выражение равно true, и для этого нам не нужно определять значение второго подвыражения.

ж) false. Обратите внимание на то, что выражение е) входит в состав выражения ж) и вычисляется по сокращенной схеме, описанной в предыдущем пункте. Полное выражение ж) эквивалентно следующему:

```
!((true || (x < y)) && true)
```

а оно, в свою очередь, эквивалентно выражению !(true && true), которое эквивалентно выражению !(true), равному результирующему значению false.

з) При вычислении этого выражения происходит ошибка, поскольку первое подвыражение, ((limit/count) > 7), включает деление на нуль.

и) true. Поскольку значением первого подвыражения (limit < 20) является true, все выражение тоже равно true, и второе подвыражение вычислять незачем. Таким образом, второе подвыражение, ((limit/count) > 7), никогда не вычисляется, и тот факт, что в нем выполняется деление на нуль, остается компьютером незамеченным. Таково одно из преимуществ выполняемого C++ сокращенного вычисления выражений.

к) При вычислении этого выражения происходит ошибка, так как первое подвыражение, ((limit/count) > 7), включает деление на нуль.

л) false. Поскольку значением первого подвыражения (limit < 0) является false, все выражение тоже равно false, и второе подвыражение вычислять незачем.

м) Если вы считаете, что это выражение бессмысленно, то вы правы. С точки зрения естественной человеческой логики это действительно так, но C++ не интересует его смысл. Он просто преобразует значения типа int в значения типа bool и затем выполняет операторы && и !. Таким образом вычисляется эта чепуха. Вспомните, что в C++ любое ненулевое целое число преобразуется в true, а значение 0 — в false. Поэтому выражение

```
(5 && 7) + (!6)
```

вычисляется следующим образом. Сначала в выражении (5 && 7) значения 5 и 7 преобразуются в true. Значение выражения true && true, равное true, преобразуется в 1. В выражении (!6) число 6 преобразуется в true, и значение выражения (!true), равное false, — в 0. Затем вычисляется выражение 1+0 и получается окончательный результат, число 1. C++ преобразует его в true, хотя, пожалуй, лучше было бы сказать, что ответ равен 1.

Нет никакой необходимости тренироваться в вычислении таких бессмысленных выражений, но все же следует понимать, как они работают, чтобы исправлять случайно допущенные ошибки, не замечаемые компилятором.

2. До сих пор мы с вами изучали операторы ветвления, итерационные операторы и операторы вызова функций.

   Примерами операторов ветвления являются `if` и `if...else`, а примерами итерационных операторов — `while` и `do...while`.

3. Выражение `2 < x < 3` допустимо, но оно не означает `(2 < x)&&(x < 3)`, как предположил бы человек, не знакомый с языком C++. На самом деле оно означает `(2 < x) < 3`. Поскольку `(2 < x)` — логическое выражение, его значением является либо `true`, либо `false`, преобразуемое в `1` или `0` соответственно, так что выражение `(2 < x) < 3` всегда равно `true`. Оно истинно независимо от значения переменной `x`.

4. Нет. Логическое выражение `j > 0` равно `false` (так как переменной `j` присвоено значение `–1`). Оператор `&&` выполняется по сокращенной схеме, и в том случае, если первое подвыражение оказывается ложным, второе не вычисляется. Первое подвыражение действительно оказывается ложным, и все выражение оценивается как `false` без вычисления второго.

5. ```
   bool in_order(int n1, int n2, int n3)
   {
   return ((n1 <= n2) && (n2 <= n3));
   }
   ```

6. ```
   bool even(int n)
   {
   return ((n % 2) == 0);
   }
   ```

7. ```
   bool is_digit(char ch)
   {
   return ('0' <= ch) && (ch <= '9');
   }
   ```

8. ```
   bool is_root_of(int root_candidate, int number)
   {
   return (number == root_candidate*root_candidate);
   }
   ```

9. ```
   Start
   Hello from the second if.
   End
   Start again
   End again
   ```

10. ```
    large
    ```

11. ```
    small
    ```

12. ```
    medium
    ```

13. ```
    Start
    Second Output
    End
    ```

14. От замены условия `(x > 10)` условием `(x > 100)` оператор `if...else` не изменится. Поэтому выходные данные будут такими же, как в упражнении 13.
    ```

15. Start
    100
    End

16. Оба варианта:

```
if (n < 0)
    cout << n << " is less than zero.\n";
else if ((0 <= n) && (n <= 100))
    cout << n << " is between 0 and 100 (inclusive).\n";
else if (n >100)
    cout << n << " is larger than 100.\n";
```

    и

```
if (n < 0)
    cout << n << " is less than zero.\n";
else if (n <= 100)
    cout << n << " is between 0 and 100 (inclusive).\n";
else
    cout << n << " is larger than 100.\n";
```

    правильны.

17. Константам типа enum по умолчанию присваиваются значения, начиная с нуля, если не задано иное. Значение каждой следующей константы на единицу больше предыдущей. Поэтому данный оператор выведет следующее:

    3 2 1 0

18. Константам типа enum по умолчанию присваиваются значения, начиная с нуля, если не задано иное. Значение каждой следующей константы на единицу больше предыдущей. Поэтому данный оператор выведет следующее:

    2 1 7 5

19. Roast worms

20. Onion ice cream

21. Chocolate ice cream
    Onion ice cream

    (Поскольку ветвь case 3 не содержит оператора break.)

22. Bon appetit!

23. 42 22

24. Мы приведем вывод непосредственно возле программного кода, чтобы было видно, что выводит каждая строка. Кроме того, немного изменим сам программный код, для того чтобы было ясно, какая переменная выводится каждым оператором. Символы <Enter> обозначают перевод строки.

```
{
    int x1 = 1;
    cout << x1 << endl;       // 1<Enter>
    {
        cout << x1 << endl;   // 1<Enter>
        int x2 = 2;
        cout << x2 << endl;   // 2<Enter>
```

```
        {
            cout << x2 << endl;   // 2<Enter>
            int x3 = 3;
            cout << x3 << endl;   // 3<Enter>
        }
        cout << x2 << endl;   // 2<Enter>
    }
    cout << x1 << endl;   // 1<Enter>
}
```

25. 2 1 0

26. 2 1

27. 1 2 3 4

28. 1 2 3

29. 2 4 6 8

30.
```
Hello 10
Hello 8
Hello 6
Hello 4
Hello 2
```

31. 2.000000 1.500000 1.000000 0.500000

32. а) цикл for;

    б) цикл while, поскольку входной список может быть пустым;

    в) то же, что в пункте б);

    г) цикл do...while, поскольку тело цикла будет выполнено хотя бы один раз.

33. а)
```
for (int i = 1; i <= 10; i++)
    if (i < 5 && i != 2)
        cout << 'X';
```

    б)
```
for (int i = 1; i <= 10; i = i + 3)
    cout << 'X';
```

    в)
```
cout << 'X';        // Необходимо, чтобы вывод был таким же.
                    // Обратите внимание на изменение инициализации переменной m.
for (long m = 200; m < 1000; m = m + 100)
    cout << 'X';
```

34. Он выведет 1024 10. Второе число является логарифмом первого числа по основанию 2.

35. Он выведет 1024 1. Точка с запятой после оператора for, вероятно, поставлена случайно.

36. Это бесконечный цикл. Рассмотрим выражение обновления управляющей переменной цикла: i = i * 2. Оно не может изменить переменную i, так как ее начальным значением является 0. Поэтому оно оставляет исходное значение переменной i, равное 0.

37. 4 3 End of Loop.

38. 4 3

39. Оператор break используется для выхода из цикла (while, do...while или for) или для выхода из оператора switch. Ни в каких других местах программы на C++ он использоваться не может. Имейте в виду, что во вложенных циклах оператор break завершает только тот цикл, в котором он расположен.

40.
```
for (int count = 1; count <= 10; count++)
    cout << "Hello\n";
```

41. В качестве сигнального значения можно использовать любое нечетное число.

```
int sum = 0, next;
cout << "Enter a list of even numbers. Place an\n"
     << "odd number at the end of the list.\n";
cin >> next;
while ((next % 2) == 0)
{
    sum = sum + next;
    cin >> next;
}
```

42. Вывод этого кода слишком длинный, чтобы поместить его здесь. Вот что он собой представляет:

```
1 * 10 = 10
1 * 9 = 9
    .
    .
    .
1 * 1 = 1
2 * 10 = 20
2 * 9 = 18
    .
    .
    .
2 * 1 = 2
3 * 10 = 30
    .
    .
    .
```

43. Трассировка переменной — это возможность наблюдать за изменениями ее значения в ходе выполнения программы. Ее можно выполнить с помощью специальных средств отладки или путем вставки в программу временных операторов вывода.

44. Если тело цикла выполняется на один раз меньше или больше, чем нужно, это называется ошибкой на единицу.

45. Ошибки на единицу распространены не только при написании циклов. Типичное рассуждение недостаточно внимательного человека таково:

    10 кольев = 100 футов изгороди/10 футов между кольями

    И последние 10 футов изгороди останутся без опоры. На самом деле нужно 11 кольев: между ними на протяжении 100 футов будет расположено 10 десятифутовых интервалов.

Практические задания

1. Напишите программу для игры «камень-бумага-ножницы». Каждый из двух пользователей вводит либо `R` (камень), либо `P` (бумага), либо `S` (ножницы). Затем программа объявляет победителя. А правила определения победителя просты: «Бумага накрывает камень, камень ломает ножницы, ножницы режут бумагу» либо «Никто не выигрывает». Не забудьте разрешить пользователям вводить буквы не только в верхнем, но и в нижнем регистре. Программа должна включать цикл, который позволит пользователю играть до тех пор, пока не надоест.
2. Напишите программу, вычисляющую начисленный процент, общую сумму на счету и минимальный взнос для возобновляемого кредитного счета. Программа вводит значение исходного баланса счета и прибавляет к нему начисленный процент, для того чтобы получить общую сумму к выплате. Процентные ставки таковы: на первую $1000 начисляется 1,5 %, а на остальную часть суммы – 1 %. Минимальный взнос равен общей сумме к выплате, если она не больше $10; в противном случае это большее из двух значений: $10 либо 10 % от общей суммы кредита. Программа должна содержать цикл, позволяющий пользователю повторять вычисления, пока он не укажет, что хочет завершить работу.
3. Напишите астрологическую программу. Пользователь вводит день своего рождения, а программа выводит его знак зодиака и гороскоп. Месяц может быть введен как число от 1 до 12. Гороскопы и даты для каждого знака можно взять из газеты. Дополните программу, чтобы в случае, если дата рождения отстоит от смежного знака на один-два дня, программа объявляла, что дата находится на границе знаков, и выводила гороскоп также для смежного знака. Эта программа будет содержать ветвление с множеством ветвей. Гороскопы храните в файлах. Программа должна содержать цикл, позволяющий пользователю повторять вычисления произвольное количество раз.
4. Напишите программу, вычисляющую стоимость телефонного разговора, которая определяется согласно следующим правилам:
   а) звонок с 8:00 до 18:00 с понедельника по пятницу оплачивается по цене, составляющей 40 центов за минуту;
   б) звонок до 8:00 или после 18:00 с понедельника по пятницу оплачивается по цене 25 центов за минуту;
   в) звонок в субботу или воскресенье оплачивается по цене 15 центов за одну минуту.

   Программа вводит значения дня недели, времени звонка, длительности соединения в минутах и стоимости разговора. Значение времени вводится в 24-часовом формате. День недели считывается как пара символов, представляющих его сокращенное название, и помещается в две переменные типа `char`. Программа должна позволять вводить буквы как в верхнем, так и в нижнем регистре. Количество минут представляет собой значение типа `int`. Предполагается, что пользователь будет округлять время разговора до целого количества

минут. Программа должна содержать цикл, позволяющий пользователю выполнять вычисления произвольное количество раз.

После полной отладки напишите еще одну версию программы, которая считывает из файла информацию обо всех телефонных звонках за неделю и записывающую счет в другой файл. Счет должен содержать информацию о каждом звонке и его стоимости, а также общую сумму стоимости всех звонков. В выходном файле звонки должны быть перечислены в том же порядке, в каком они заданы во входном файле. Если это задание выполняется в классе, спросите у преподавателя имена файлов.

5. Напишите программу, вводящую номер года, записанный арабскими цифрами как четырехзначное число, и выводящую его римскими цифрами. Важные для этой задачи римские цифры таковы: I соответствует 1, V – 5, X – 10, L – 50, C – 100, D – 500 и M – 1000. Напомним, что некоторые числа формируются путем вычитания одной или нескольких «единиц» из ближайшего числа, для которого имеется цифра. Например, IV соответствует 4 и формируется как V минус I, XL соответствует 40, CM – 900 и т. д. Вот несколько примеров значений года: MCM обозначает 1900, MCML – 1950, MCMLX – 1960, MCMXL – 1940, MCMLXXXIX – 1989. Предполагается, что значение года находится в пределах между 1000 и 3000. Программа должна содержать цикл, позволяющий пользователю повторять эти вычисления произвольное количество раз.

6. Напишите программу, подсчитывающую количество очков на руках у игрока в двадцать один. В этой игре участник получает от двух до пяти карт. (Он сам решает, сколько именно, но в данном упражнении это не имеет значения.) Карты с двойки по десятку оцениваются соответствующим их названию количеством очков. Валет, дама и король оцениваются в 10 очков. Цель игры заключается в том, чтобы максимально приблизиться к 21 очку, но не превысить это значение. Набравший свыше 21 очка теряет все свои очки. Туз оценивается либо в 1, либо в 11 очков, по желанию игрока. Например, туз и 10 могут оцениваться как 11 или 21 очко. Поскольку второй вариант лучше, этот набор оценивается в 21. Туз и две восьмерки можно оценить как 17 или 27. Поскольку 27 сделает игрока банкротом, выбирается значение 17.

   У игрока спрашивают, сколько карт он хочет получить, и он отвечает 2, 3, 4 или 5. Затем у него узнают значения карт. Значениями карт являются числа от 2 до 10, валет, дама, король и туз. Удобным способом обработки ввода является использование типа `char`, чтобы значение двойка считывалось как `2`. Значения от двойки до девятки вводятся как символы от `2` до `9`. Входные значения десятка, валет, дама, король и туз вводятся как символы `t`, `j`, `q`, `k` и `a`. Программа должна допускать использование символов верхнего и нижнего регистров.

   После ввода всех значений программа должна преобразовать их в очки, соответствующие каждой карте. Особого внимания требуют тузы. Затем программа выводит значение от 2 до 21 (включительно) или слово `Busted` (банкрот). Пользуйтесь функциями. Скорее всего, в программе будет одно или больше

многонаправленных ветвлений, реализованных с помощью оператора `switch` или `if...else`. Программа должна включать цикл, позволяющий пользователю выполнять вычисления произвольное количество раз.

7. Процент по ссуде начисляется на текущий, с каждым разом уменьшающийся баланс, и поэтому ссуда со ставкой в 14 % будет значительно меньше 14 % исходного баланса. Напишите программу, принимающую значения суммы ссуды и процентной ставки и выводящую значения месячного взноса и баланса вплоть до погашения ссуды. Предполагается, что ежемесячно выплачивается двенадцатая часть исходной суммы ссуды и что процент начисляется, исходя из уменьшающегося баланса. Таким образом, для ссуды в $20 000 месячный взнос составляет $1000. Если процентная ставка равна 10 %, то каждый месяц начисляется одна двенадцатая часть от 10 % оставшегося баланса. Поэтому за первый месяц в качестве процентов должно быть начислено (10 % от $20 000)/12, то есть $166,67, так что после выплаты взноса остаток баланса составит $19 166,67. А в следующем месяце будет начислен процент (10 % от $19 166,67)/12, и т. д. Кроме того, программа должна выводить общую сумму процентных начислений за все время погашения ссуды.

   Напоследок определите простой годовой процент, начисленный в итоге на исходный баланс. Например, когда по $10 000 ссуде в течение двух лет была выплачена $1000 процентов, годовой процент составляет $500, то есть 5 % от ссуды в $10 000. При выполнении этого упражнения вы можете сами решить, каким образом следует осуществлять ввод-вывод: в интерактивном режиме или с помощью файлов. Если данные будут вводиться посредством клавиатуры и выводится на экран, программа должна позволять пользователю повторять вычисления произвольное количество раз.

8. Числа Фибоначчи F_n определяются так: F_0 равно 1, F_1 равно 1, а число

   $$F_{i+2} = F_i + F_{i+1}$$

   где $i = 0, 1, 2, \ldots$. Иными словами, каждое следующее число является суммой двух предыдущих. Эти числа используются, в частности, при оценке роста популяции. При отсутствии смертей данная последовательность показывает размер популяции через заданные периоды. Период составляет половину времени, необходимого для достижения организмом репродуктивного возраста. По достижении такого возраста организм воспроизводится один раз за период. Эту формулу можно применять для популяции с бесполым размножением.

   Предположим, что популяция зеленой плесени растет с указанной скоростью и период ее размножения — 5 дней. Начальный вес популяции — 10 фунтов, и спустя 5 дней она все еще весит 10 фунтов. Через 10 дней ее вес составляет уже 20 фунтов, через 15 дней — 30 фунтов, через 20 дней — 50 фунтов и т. д. Напишите программу, принимающую значения начального веса популяции и срока ее размножения и выводящую значение ее веса по истечении заданного срока. Предполагается, что размер популяции в течение четырех дней остается неизменным, а на пятый день увеличивается. Программа позволяет пользователю повторять вычисления сколько угодно раз.

Альтернативная версия программы с использованием файлов считывает входные данные из файла и записывает результат в другой файл. Из входного файла она считывает значения начального веса популяции и срока ее размножения, а в выходной записывает значения начального веса популяции, срока ее размножения и веса популяции по истечении заданного срока. Входной файл должен содержать по два числа в строке (значения начального веса популяции и срока ее размножения). В программе будет использоваться цикл, обрабатывающий все имеющиеся в файле данные.

9. Значение e^x можно представить в виде следующей суммы:

   $1 + x + x^2/2! + x^3/3! + ... + x^n/n!$

   Напишите программу, принимающую значение x и выводящую сумму для каждого значения n из диапазона от 1 до 100. Кроме того, программа должна вывести значение e^x, вычисленное с помощью предопределенной функции `exp`. Такая функция, вызываемая с одним аргументом, например, `exp(x)`, возвращает приблизительное значение e^x. Она находится в заголовочном файле `cmath`. Программа должна позволять пользователю повторять вычисления с новыми значениями x произвольное количество раз.

   Для хранения факториалов пользуйтесь переменными типа `double` — иначе вы рискуете вызвать переполнение целочисленных переменных (или же организуйте вычисления так, чтобы избежать непосредственного вычисления факториала). Программа может вводить данные с клавиатуры или из файла и выводить результаты в файл. Если данные вводятся из файла, задайте имя входного файла.

Глава 8 Дружественные функции и перегрузка операторов

Дайте нам инструменты, и мы закончим работу.

Уинстон Черчилль

В настоящей главе описаны дополнительные средства определения функций и операторов для классов. Здесь мы подробно расскажем каким образом производится перегрузка обычных операторов (+, * и /), для того чтобы их можно было использовать с определяемыми программистом классами точно так же, как с предопределенными типами.

8.1. Дружественные функции

Доверяйте своим друзьям.

Распространенный совет

До сих пор мы реализовывали операции класса, такие как ввод, вывод, доступ к переменным-членам и т. п., в виде функций-членов класса, но некоторые из них лучше реализовать в виде обычных функций (не членов класса). В этом разделе мы поговорим о технологии определения операций над объектами вне класса, к которому они относятся. Начнем с простого примера.

Пример: функция равенства

В главе 6 нами был создан класс `DayOfYear`, представляющий дату, например 1 января или 4 июля, для работы с информацией о праздниках, днях рождения или других ежегодных событиях. Мы постепенно усовершенствовали этот класс, а его

последнюю версию предложили вам разработать самостоятельно (упражнение для самопроверки 23 главы 6). В листинге 8.1 эта версия класса DayOfYear приведена еще раз с одним дополнением — функцией equal, проверяющей, представляют ли два объекта класса DayOfYear одну и ту же дату.

Листинг 8.1. Функция для сравнения объектов

```
// Программа, демонстрирующая функцию equal. Класс DayOfYear точно такой же,
// как в упражнениях для самопроверки 23 и 24 главы 6.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
    DayOfYear(int the_month, int the_day);
    // Предусловие: значения аргументов the_month и the_day составляют
    // допустимую дату. Устанавливает дату в соответствии
    // со значениями аргументов.

    DayOfYear();
    // Инициализирует дату значением 1 января.

    void input();

    void output();

    int get_month();
    // Возвращает значение месяца: 1 соответствует январю, 2 — февралю и т. д.

    int get_day();
    // Возвращает номер дня месяца.
private:
    void check_date();
    int month;
    int day;
};

bool equal(DayOfYear date1, DayOfYear date2);
// Предусловие: аргументы date1 и date2 содержат значения.
// Возвращает true, если значения аргументов date1 и date2 представляют
// одну и ту же дату; в противном случае возвращает false.

int main()
{
    DayOfYear today, bach_birthday(3, 21);

    cout << "Enter today's date:\n";
    today.input();
    cout << "Today's date is ";
    today.output();

    cout << "J. S. Bach's birthday is ";
    bach_birthday.output();

    if ( equal(today, bach_birthday))
        cout << "Happy Birthday Johann Sebastian!\n";
    else
```

продолжение

Листинг 8.1 (*продолжение*)

```
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}

bool equal(DayOfYear date1, DayOfYear date2)
{
     return (date1.get_month() == date2.get_month() &&
               date1.get_day() == date2.get_day());
}

DayOfYear::DayOfYear(int the_month, int the_day)
                  : month(the_month), day(the_day)
{
    check_date();
}

DayOfYear::DayOfYear() : month(1), day(1)
{
    // Тело намеренно оставлено пустым.
}

void DayOfYear::check_date()
{
    if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
    {
        cout << "Illegal date.\n";
        exit(1);   // Функция exit завершает работу программы.
    }
}
int DayOfYear::get_month()
{
    return month;
}

int DayOfYear::get_day()
{
    return day;
}

// Используем класс iostream:
void DayOfYear::input()
{
    cout << "Enter the month as a number: ";
    cin >> month;
    check_date();
    cout << "Enter the day of the month: ";
    cin >> day;
    check_date();
}

// Используем класс iostream.
void DayOfYear::output()
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

Пример диалога

```
Enter today's date:
Enter the month as a number: 3
Enter the day of the month: 21
Today's date is month = 3, day = 21
J. S. Bach's birthday is month = 3, day = 21
Happy Birthday Johann Sebastian!
```

Предположим, что today и bach_birthday — два объекта типа DayOfYear и им присвоены значения, представляющие некоторые даты. Для того чтобы узнать, представляют ли они одну и ту же дату, нужно проверить следующее логическое выражение:

```
equal(today, bach_birthday)
```

Функция equal возвращает true, если значения аргументов today и bach_birthday представляют одну и ту же дату. В программе листинга 8.1 данное логическое выражение используется для управления оператором if...else.

Названная функция определена очень просто — две даты равны, если они представляют один и тот же месяц и один и тот же день этого месяца. Для сравнения месяцев и дней, представляемых двумя объектами, используются аксессоры get_month и get_day.

Обратите внимание, что мы не стали делать функцию equal членом класса DayOfYear. Ее можно было бы включить в класс, но она сравнивает два объекта этого класса, и если сделать ее функцией-членом, придется решать, какую дату — первую или вторую — будет представлять вызывающий объект. Вместо того чтобы выбрать одну из дат, мы определили equal как обычную независимую функцию, принимающую в качестве аргументов два объекта, представляющих даты.

Упражнение для самопроверки

1. Напишите определение функции before, принимающей два аргумента типа DayOfYear, определенного в программе листинга 8.1. Функция возвращает значение типа bool, равное true, когда первый аргумент представляет более раннюю дату, чем второй; в противном случае функция возвращает false. Например, дата 5 января более ранняя, чем дата 2 февраля.

Применение дружественных функций

Если класс содержит полный набор функций-аксессоров, на их основе можно определить функцию, которая сравнивает два объекта или выполняет другие вычисления, зависящие от закрытых переменных-членов. Однако такая методика, хоть и обеспечивает функции доступ к закрытым переменным-членам, все же недостаточно эффективна. Давайте вновь обратимся к определению функции equal в листинге 8.1. Чтобы прочитать номер месяца, эта функция должна вызвать аксессор get_month, а для прочтения номера дня — аксессор get_day. Данный код работает и выполняет свою задачу, но он будет проще и эффективнее, если обеспечить возможность его непосредственного обращения к переменным-членам объектов.

Усовершенствованное подобным образом определение функции equal, приведенной в листинге 8.1, может быть таким:

```
bool equal(DayOfYear date1, DayOfYear date2)
{
    return ( date1.month == date2.month &&
             date1.day == date2.day );
}
```

У этого определения есть только один недостаток — оно недопустимо, поскольку переменные month и day являются закрытыми членами класса DayOfYear. Закрытые переменные-члены (как и закрытые функции-члены) обычно недоступны в теле функций, не являющихся членами их класса, а equal не принадлежит к членам класса DayOfYear. Однако существует возможность открыть функции, не являющейся членом класса, доступ к его закрытым переменным-членам. Она заключается в том, чтобы сделать функцию equal дружественной функцией класса DayOfYear.

Дружественная функция класса не является его членом, но имеет такой же доступ к его закрытым переменным-членам, как функции-члены данного класса. Это означает, что она может непосредственно считывать и даже устанавливать их значения. Для того чтобы сделать функцию дружественной функцией класса, нужно соответствующим образом объявить ее в определении класса. Например, в программе листинга 8.2 в определение класса DayOfYear добавлено объявление дружественной функции equal, которое включается в список объявлений функций в определении класса и предваряется ключевым словом friend.

Дружественная функция добавляется в определение класса точно так же, как любая функция-член, то есть ее объявление включается в список объявлений функций в определении класса, но в отличие от функций-членов, перед ее объявлением ставится ключевое слово friend. Однако, хотя дружественная функция объявляется в определении класса наряду с функциями-членами, она не является членом этого класса; это обычая функция со специальными правами доступа к переменным-членам класса. Дружественная функция определяется и вызывается как обычная функция. В частности, заголовок определения функции equal в программе листинга 8.2 не содержит спецификатор DayOfYear::. Вызов этой функции выполняется без оператора точка. Она принимает в качестве аргумента объект типа DayOfYear, подобно тому как любая функция, не являющаяся членом класса, может принимать аргументы любого типа. Но дружественная функция может обращаться к закрытым переменным-членам и закрытым функциям-членам класса по именам, то есть имеет те же права доступа, что и функции-члены.

Листинг 8.2. Функция, выполняющая сравнение объектов, объявлена как дружественная функция класса

```
// Программа, демонстрирующая использование функции equal. В этой версии
// функция equal является дружественной функцией класса DayOfYear.
#include <iostream>
using namespace std;

class DayOfYear
{
public:
friend bool equal(DayOfYear date1, DayOfYear date2);
```

```
    // Предусловие: аргументы date1 и date2 имеют значения.
    // Возвращает true, если значения аргументов date1 и date2 представляют
    // одну и ту же дату, в противном случае возвращает false.

    DayOfYear(int the_month, int the_day);
    // Предусловие: значения аргументов the_month и the_day составляют
    // допустимую дату. Устанавливает дату в соответствии
    // со значениями аргументов.

    DayOfYear();
    // Инициализирует дату значением 1 января.

    void input();

    void output();

    int get_month();
    // Возвращает номер месяца: 1 для января, 2 для февраля и т. д.

    int get_day();
    // Возвращает номер дня месяца.
private:
    void check_date();
    int month;
    int day;
};

int main()
{
DayOfYear today, bach_birthday(3, 21);

    cout << "Enter today's date:\n";
    today.input();
    cout << "Today's date is ";
    today.output();

    cout << "J. S. Bach's birthday is ";
    bach_birthday.output();

    if ( equal(today, bach_birthday))
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}

bool equal(DayOfYear date1, DayOfYear date2)   // К закрытым переменным-членам month и day
                                               // можно обращаться по имени.
{
    return (date1.month == date2.month &&
                           date1.day == date2.day);
}

DayOfYear::DayOfYear(int the_month, int the_day)
                  : month(the_month), day(the_day)
{
    check_date();
}
```

продолжение ⇩

Листинг 8.2 (*продолжение*)

```
DayOfYear::DayOfYear() : month(1), day(1)
{
    // Тело намеренно оставлено пустым.
}

void DayOfYear::check_date()
{
    if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
    {
        cout << "Illegall date.\n";
        exit(1);   // Функция exit завершает работу программы.
    }
}
int DayOfYear::get_month()
{
    return month;
}

int DayOfYear::get_day()
{
    return day;
}

// Используем библиотеку классов iostream.
void DayOfYear::input()
{
    cout << "Enter the month as a number: ";
    cin >> month;
    check_date();
    cout << "Enter the day of the month: ";
    cin >> day;
    check_date();
}

// Используем библиотеку классов iostream.
void DayOfYear::output()
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

Совет программисту: определяйте и аксессоры, и дружественные функции

Может показаться, что если все ключевые функции программы сделать дружественными функциями класса, отпадет необходимость в аксессорах и мутаторах. Поскольку дружественные функции имеют доступ к закрытым переменным-членам класса, зачем нужны еще и специальные функции-члены для доступа к тем же переменным? В этом есть доля истины. Верно, что если все функции программы будут дружественными функциями класса, который в ней используется, аксессоры и мутаторы окажутся не у дел. Но суть проблемы в том, что не следует делать дружественными все функции программы.

Давайте еще раз вернемся к определению класса DayOfYear, приведенному в листинге 8.2. Этот класс можно использовать в другой программе, которой тоже возможно потребуется доступ к значению месяца объекта DayOfYear. Например, программа может вычислять, сколько месяцев остается от заданного месяца до конца года. Для этого в функции main будет содержаться такой код:

```
DayOfYear today;
cout << "enter today's date: \n";
today.input();
cout << "There are " << (12 - today.get_month())
     << " months left in this year.\n";
```

Вызов функции today.get_month() никак нельзя заменить обращением today.month, поскольку month является закрытой переменной класса DayOfYear. Это означает, что класс обязательно должен содержать аксессор.

Теперь понятно, почему каждый класс должен содержать аксессор. То же самое относится и к мутаторам. Итак, вопрос с аксессорами и мутаторами решен в их пользу, и сразу возникает противоположный вопрос: а нужны ли вообще в таком случае дружественные функции класса? Ответ на него неоднозначен. Заметьте, что функцию equal из нашего примера можно определить либо как дружественную, и не пользоваться аксессорами (как в листинге 8.2), либо не как дружественную, и применять аксессоры (как в листинге 8.1). В большинстве случаев единственной причиной, по которой функцию определяют как дружественную, является желание упростить и сделать более эффективным ее определение.

Дружественная функция

Дружественная функция класса — это обычная функция, которой предоставлен доступ к закрытым членам объектов данного класса. Для того чтобы сделать функцию дружественной функцией класса нужно добавить ее объявление в список объявлений функций в определении класса и предварить его ключевым словом friend. Объявление дружественной функции можно поместить либо в открытую, либо в закрытую секцию определения класса. Но поскольку эта функция в любом случае будет открытой, логичнее объявить ее в открытой секции.

Синтаксис

```
class имя_класса
{
public:
    friend объявление_дружественной_функции_1
    friend объявление_дружественной_функции_2
// Дружественные функции не обязательно перечислять первыми.
// Их объявления можно чередовать с объявлениями других функций.
                    .
                    .
                    .
    объявления_функций_членов
private:
    объявления_закрытых_членов
};
```

продолжение

Пример

```
class FuelTank
{
public:
    friend double need_to_fill(FuelTank tank);
    // Предусловие: переменные-члены объекта tank
    // содержат значения.
    // Возвращает значение количества литров бензина, необходимое
    // для заполнения бака, представленного объектом tank.
    FuelTank(double the_capacity, double the_level);

    FuelTank();
    void input();
    void output();
    private:
    double capacity;   // В литрах.
    double level;
};
```

Дружественная функция класса не является его членом. Она определяется и вызывается как обычная функция. В ее вызове не требуются оператор точка и ссылка на объект, а в определении не нужен спецификатор типа.

Совет программисту: используйте и функции-члены, и обычные функции

Роли функций-членов и дружественных функций класса очень сходны. Фактически иногда даже трудно решить, что лучше: сделать некоторую функцию членом класса или дружественной функцией. В большинстве случаев можно прибегнуть к любому решению, и вне зависимости от выбора функция будет выполнять ту же задачу и тем же способом. Но существуют ситуации, в которых необходима именно функция-член, и другие ситуации, когда больше подойдет дружественная функция (или даже обыкновенная функция, которая не является дружественной, как функция `equal` в программе листинга 8.1). При выборе решения имейте ввиду следующее:

- функция-член класса больше подходит для задачи, связанной с единственным объектом;
- функция, не являющаяся членом класса, предпочтительнее для операций над несколькими объектами. Например, функция `equal` в программе, приведенной в листингах 8.1 и 8.2, работает с двумя объектами, поэтому мы сделали ее внешней по отношению к классу (в данном случае дружественной).

А вот при выборе между дружественной и обычной функцией с использованием аксессоров и мутаторов исходят из соображений эффективности и личных предпочтений программиста. Если класс содержит достаточно аксессоров и мутаторов, то годятся оба подхода.

Приведенные правила выбора между функцией-членом и внешней по отношению к классу функцией не полностью охватывают проблему. По мере накопления опыта вы обнаружите ситуации, когда их можно нарушать. Более точным,

хотя и менее понятным правилом является следующее: функции-члены лучше использовать для задач, тесно связанных с одним объектом, а внешние функции — для задач, в которых участвуют несколько объектов. Однако это правило не является достаточно определенным, и неопытным программистам проще полагаться на простое правило, приведенное выше.

Пример: класс Money

В листинге 8.3 приведено определение класса Money, представляющее денежную сумму в долларах США. Значение этого класса реализовано в виде одного целочисленного значения, представляющего сумму в центах. Например, сумма $9,95 хранится как 995. Поскольку мы используем для представления денежной суммы целое число, сумма представлена точно, а не приблизительно, как было бы при использовании типа данных double.

Листинг 8.3. Класс Money

```
// Программа, в которой используется класс Money.
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

class Money
{
public:
    friend Money add(Money amount1, Money amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает сумму значений аргументов amount1 и amount2.

    friend bool equal(Money amount1, Money amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает true, если аргументы amount1 и amount2 содержат
    // одинаковые значения; в противном случае возвращает false.

    Money(long dollars, int cents);
    // Инициализирует объект таким образом, чтобы он представлял
    // заданное в аргументах количество долларов и центов.
    // Если сумма отрицательна, то значения аргументов dollars
    // и cents тоже должны быть отрицательными.

    Money(long dollars);
    // Инициализирует объект таким образом,
    // чтобы он представлял значение $dollars.00.

    Money();
    // Инициализирует объект таким образом,
    // чтобы он представлял значение $0.00.

    double get_value();
    // Предусловие: вызывающему объекту присвоено значение.
    // Возвращает значение денежной суммы, записанное в вызывающем объекте.
```

продолжение

Листинг 8.3 (*продолжение*)

```
    void input(istream& ins);
    // Предусловие: если ins – входной файловый поток, то он уже соединен
    // с файлом. В этот поток введено значение денежной суммы включая символ доллара.
    // Отрицательные значения сумм вводятся как -$100.00.
    // Постусловие: вызывающему объекту присвоено значение денежной суммы,
    // прочитанное из входного потока insurance.

    void output(ostream& outs);
    // Предусловие: если outs – выходной файловый поток, то он уже соединен с файлом.
    // Постусловие: символ доллара и значение денежной суммы, хранящееся
    // в вызывающем объекте, записаны в выходной поток outs.
private:
    long all_cents;
};

int digit_to_int(char c);
// Используется в определении функции Money::input.
// Предусловие: c – одна из цифр от '0' до '9'.
// Возвращает целочисленное значение этого символа;
// например, digit_to_int('3') возвращает 3.

int main()
{
    Money your_amount, my_amount(10, 9), our_amount;
    cout << "Enter an amount of money: ";
    your_amount.input(cin);
    cout << "Your amount is ";
    your_amount.output(cout);
    cout << endl;
    cout << "My amount is ";
    my_amount.output(cout);
    cout << endl;

    if (equal(your_amount, my_amount))
        cout << "We have the same amounts.\n";
    else
        cout << "One of us is richer.\n";
    our_amount = add(your_amount, my_amount);
    your_amount.output(cout);
    cout << " + ";
    my_amount.output(cout);
    cout << " equals ";
    our_amount.output(cout);
    cout << endl;
    return 0;
}

Money add(Money amount1, Money amount2)
{
    Money temp;

    temp.all_cents = amount1.all_cents + amount2.all_cents;
    return temp;
}

bool equal(Money amount1, Money amount2)
```

```
{
    return (amount1.all_cents == amount2.all_cents);
}

Money::Money(long dollars, int cents)
{
    // Если одно из чисел отрицательно, а другое положительно.
    if(dollars*cents < 0)
    {
        cout << "Illegal values for dollars and cents.\n";
        exit(1);
    }
    all_cents = dollars*100 + cents;
}

Money::Money(long dollars) : all_cents(dollars*100)
{
    // Тело намеренно оставлено пустым.
}

Money::Money() : all_cents(0)
{
    // Тело намеренно оставлено пустым.
}

double Money::get_value()
{
    return (all_cents * 0.01);
}

// Используем библиотеки классов iostream, cctype, cstdlib.
void Money::input(istream& ins)
{
    char one_char, decimal_point,
         digit1, digit2;   // Цифры, определяющие количество центов.
    long dollars;
    int cents;
    bool negative;   // Устанавливается в true, если введено значение отрицательной
                     // денежной суммы.

    ins >> one_char;
    if (one_char == '-')
    {
        negative = true;
        ins >> one_char;  // Считывает '$'.
    }
    else
        negative = false;
    // Если введено допустимое значение, one_char == '$'.

    ins >> dollars >> decimal_point >> digit1 >> digit2;

    if (one_char != '$' || decimal_point != '.'
         || !isdigit(digit1) || !isdigit(digit2))
    {
        cout << "Error illegal form for money input\n";
        exit(1);
    }
```

продолжение

Листинг 8.3 (*продолжение*)

```
    cents = digit_to_int(digit1)*10 + digit_to_int(digit2);
    all_cents = dollars*100 + cents;
    if (negative)
        all_cents = -all_cents;
}

// Используем библиотеки классов cstdlib и iostream.
void Money::output(ostream& outs)
{
    long positive_cents, dollars, cents;
    positive_cents = labs(all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (all_cents < 0)
        outs << "-$" << dollars << '.';
    else
        outs << "$" << dollars << '.';

    if (cents < 10)
        outs << '0';
    outs << cents;
}

int digit_to_int(char c)
{
    return ( int(c) - int('0') );
}
```

Пример диалога

```
Enter an amount of money: $123.45
Your amount is $123.45
My amount is $10.09
One of us is richer.
$123.45 + $10.09 equals $133.54
```

Данное значение, представляющее сумму в центах, хранится в переменной-члене `all_cents`. Мы могли бы использовать для нее тип данных `int`, но некоторые компиляторы слишком сильно ограничивают значения этого типа. В некоторых реализациях C++ для хранения значений типа `int` выделяется всего 2 байта. Это означает, что наибольшее значение, которое можно хранить в переменной типа `int`, лишь немногим превышает 32 000, а поскольку 32 000 центов — это лишь $320, такой тип данных нам явно не подходит. Для того чтобы в объектах нашего класса могли храниться суммы, значительно превышающие $320, мы назначим переменной-члену `all_cents` тип данных `long`. Те компиляторы, которые выделяют для типа данных `int` 2 байта, реализуют тип `long` как 4 байта. Тип данных `long` — такой же целочисленный тип, как и `int`, отличающийся лишь тем, что его максимальное значение намного больше — в большинстве систем оно составляет два миллиарда или более. (Тип данных `long` называется также `long int`, оба эти имени соответствуют одному и тому же типу данных.)

Для класса `Money` определены две операции, реализованные в виде дружественных функций: `add` и `equal` (см. листинг 8.3). Первая из них возвращает объект `Money`,

значением которого является сумма значений двух других объектов Money, передаваемых этой функции в качестве аргументов. Вторая функция сравнивает два объекта Money: вызов equal(amount1, amount2) возвращает значение true, если объекты amount1 и amount2 представляют равные денежные суммы.

Обратите внимание, что класс Money считывает и записывает денежные суммы в привычном формате, таком как $9,95 или -$9,95. Рассмотрим функцию-член input (также приведенную в листинге 8.3), выполняющую ввод значения суммы. Сначала она считывает один символ, которым может быть либо символ доллара ('$'), либо знак минус ('-'). Во втором случае функция запоминает, что вводится отрицательное значение суммы, для чего присваивает переменной negative значение true. Затем она считывает второй символ, которым должен быть символ доллара. Если же первый символ не является знаком минус, переменной negative присваивается значение false. К этому моменту прочитан знак минус, если он имеется, и символ доллара. Далее функция считывает количество долларов как значение типа long и помещает его в локальную переменную dollars. Прочитав долларовую часть входного значения, функция считывает оставшуюся часть как значения типа char — это три символа (десятичный разделитель и две цифры).

У вас может возникнуть искушение прочитать десятичный разделитель как символ, а две следующие за ним цифры как значение типа int. Но так поступать не следует, поскольку некоторые компиляторы C++ не читают числа с ведущими нулями так, как вам бы хотелось (подробно об этом рассказано в разделе «Ловушка: ведущие нули в числовых константах»), и если при вводе суммы вроде $7,09 отдельно считывать число 09, C++ может интерпретировать его неправильно.

Следующий оператор присваивания преобразует две цифры, составляющие количество центов, в одно целое число, хранящееся в локальной переменной cents:

```
cents = digit_to_int(digit1)*10 + digit_to_int(digit2);
```

После выполнения такого присваивания значением этой переменной станет количество центов во входной сумме.

Вспомогательная функция digit_to_int принимает в качестве аргумента одну цифру, такую как '3', и преобразует ее в соответствующее значение типа int, в данном случае 3. Вспомогательная функция нужна нам потому, что функция-член input вводит количество центов как два значения типа char и хранит их в переменных digit1 и digit2, представляющих первую и вторую цифры. Но после ввода эти цифры нужно преобразовать в число, для чего мы и пользуемся функцией digit_to_int (например, преобразуем цифру '3' в число 3). Определение функции digit_to_int приведено в листинге 8.3. Можете просто принять на веру, что она выполняет поставленную задачу, и воспринимать ее как черный ящик. Вам нужно знать только, что вызов digit_to_int('0') возвращает 0, вызов digit_to_int('1') возвращает 1 и т. д. Хотя разобраться в том, как работает эта функция, совсем не трудно, так что если хотите, прочитайте факультативный раздел с описанием ее реализации.

После того как локальным переменным dollars и cents присвоены значения количества долларов и центов, являющиеся входными значениями, определить общую сумму в центах (которую нам нужно присвоить переменной-члену all_cents) не

составляет труда. Следующий оператор присваивает переменной-члену all_cents введенное пользователем значение суммы, выраженное в центах:

```
all_cents = dollars*100 + cents;
```

Но учтите, что он всегда присваивает переменной положительное число. Если же пользователь ввел отрицательное значение, его нужно взять со знаком минус:

```
if (negative)
    all_cents = -all_cents;
```

Функция-член output (приведенная в листинге 8.3) вычисляет количество долларов и центов на основании значения переменной-члена all_cents. Для этого она просто выполняет целочисленное деление на 100 с остатком. Например, если переменная all_cents содержит значение 995 (центов), количество долларов равно 995/100, то есть 9, а количество центов равно 995%100, то есть 95. Таким образом, для значения all_cents, равного 995 (центов), выходным значением будет $9.95.

В определении функции-члена output должна быть предусмотрена возможность вывода отрицательных значений денежных сумм. Результат целочисленного деления с отрицательным аргументом не имеет стандартного определения и в разных реализациях C++ может быть различным. Для того чтобы это не вызвало проблем, нужно перед выполнением деления получить абсолютное значение переменной all_cents, которое можно вычислить с помощью стандартной функции labs. Она, подобно функции abs, возвращает абсолютное значение своего аргумента, но в качестве такового принимает значение типа long. Как и функция abs, функция labs находится в библиотеке с заголовочным файлом cstdlib. (Некоторые версии C++ не содержат функции labs, но ее легко определить самостоятельно.)

Реализация функции digit_to_int (факультативный материал)

Определение функции digit_to_int, приведенное в листинге 8.3, следующее:

```
int digit_to_int(char c)
{
    return (int(c) - int('0'));
}
```

На первый взгляд формула вычисления возвращаемого ею значения может показаться странной, но на самом деле она выполняет очень простое действие. Подлежащая преобразованию цифра, например '3', задается в параметре c, а возвращаемым значением функции должно быть соответствующее значение типа int, в данном примере 3. Как говорилось в главах 2 и 5, значения типа char в C++ реализованы как числа. К сожалению, числовой реализацией цифры '3' не является число 3. Функция преобразования типа int(c) возвращает число, реализующее символ, заданный в ее аргументе, в виде значения типа int. Таким образом, мы преобразуем символ, хранящийся в переменной c, в число типа int, но это не то число, которое нам нужно. Например, int('3') возвращает не 3, а некое другое число. Нам же нужно преобразовать цифру в обычное ее числовое значение (скажем, '3' в 3). Это числовое значение можно получить на основе значения int(c), применив к нему некоторое преобразование.

Известно, что цифры в C++ закодированы по порядку, то есть `int('0')+1` равно `int('1')`; `int('1')+1` равно `int('2')`; `int('2')+1` равно `int('3')` и т. д. Этого достаточно, чтобы получить нужное нам значение. Оно следующее:

```
int(c) - int('0')
```

Теперь посмотрим, почему это так. Если `c` содержит `'0'`, `int(c) - int('0')` равно `int('0') - int('0')`, то есть 0. Если `c` содержит `'1'`, `int(c) - int('0')` равно `int('1') - int('0')` или `(int('0')+1) - int('0')`, или `int('0') - int('0')+1`, то есть 1. Точно так же можно проверить и другие цифры от `'2'` до `'9'`; значение каждой из них на единицу больше значения предыдущей.

Ловушка: ведущие нули в числовых константах

Вот объявление объектов, взятое из функции `main` программы, приведенной в листинге 8.3:

```
Money your_amount, my_amount(10, 9), our_amount;
```

Два аргумента в объявлении `my_amount(10, 9)` представляют $10,09. Поскольку обычно центы записываются как «,09», вы, возможно, захотите записать объявление этого объекта в виде `my_amount(10, 09)`, но это может вызвать проблемы. В математике 9 и 09 представляют одно и то же число, однако некоторые компиляторы Microsoft интерпретируют ведущий нуль как признак использования другой системы счисления, так что в C++ константы 9 и 09 не обязательно представляют одно и то же число. Ведущий нуль может означать, что число записано по основанию 8, а не 10. Так как в восьмеричной системе счисления цифра 9 вообще не используется, константа 09 в C++ не имеет смысла. Константы от 00 до 07 будут восприняты, но в некоторых системах проблемы могут возникнуть и с ними.

Стандарт ANSI C++ определяет, что по умолчанию входные данные должны интерпретироваться как десятичные независимо от наличия ведущих нулей. Компилятор C++ проекта GNU, называемый g++, и компилятор VC++ Microsoft соответствуют этому стандарту, поэтому для них проблем с ведущими нулями нет. Большинство производителей также стремятся к тому, чтобы их продукты соответствовали стандартам ANSI, поэтому со временем проблема с ведущими нулями должна исчезнуть. Напишите маленькую программу, позволяющую проверить, как ведет себя ваш компилятор.

Упражнения для самопроверки

2. Чем отличается дружественная функция класса от функции-члена класса?
3. Предположим, что вы хотите добавить дружественную функцию в класс `DayOfYear`, определенный в листинге 8.2. Эта функция будет называться `after` и принимать два аргумента типа `DayOfYear`. Она возвращает значение `true`, если первый аргумент представляет более позднюю дату, чем второй; в противном случае она возвращает `false`. Например, 2 февраля — более поздняя дата, чем 5 января. Что нужно добавить в определение класса `DayOfYear`, приведенное в листинге 8.2?

4. Допустим, вы хотите включить в класс Money, определенный в листинге 8.3, дружественную функцию subtraction для вычитания денежных сумм. Что для этого следует добавить в определение класса? Функция subtraction должна принимать два аргумента типа Money и возвращать объект типа Money, значением которого является разность значений первого и второго аргументов.

5. Обратите внимание на функцию-член output в определении класса Money, приведенном в листинге 8.3. Для вывода значения типа Money на экран нужно вызвать эту функцию с аргументом cout. Например, если purse — объект типа Money, для вывода хранящегося в нем значения денежной суммы на экран нужно выполнить в программе следующий оператор:

   ```
   purse.output(cout);
   ```

   Возможно, было бы лучше, если бы для вывода значения на экран можно было вообще не задавать выходной поток.

   Перепишите определение класса Money, приведенное в листинге 8.3. Единственным изменением должна быть перегрузка функции-члена output, чтобы в новой версии класса было две таких функции. Одна из них — та, что определена в программе листинга 8.3, а вторая — без аргументов, направляющая вывод на экран. Для этой новой версии класса Money следующие два вызова будут эквивалентны:

   ```
   purse.output(cout);
   ```

   и

   ```
   purse.output();
   ```

   Очевидно, что второй из них проще. Заметьте, что поскольку первая версия функции-члена output тоже сохранена, вывод по-прежнему можно направлять в файл. Если outs — выходной файловый поток, соединенный с файлом, оператор

   ```
   purse.output(outs);
   ```

   выведет значение объекта purse в этот файл.

6. Посмотрите на определение функции-члена input класса Money, приведенное в листинге 8.3. Если пользователь вводит неправильное значение, эта функция выводит сообщение об ошибке и завершает работу программы. Так, когда пользователь забудет ввести символ доллара, функция выведет соответствующее сообщение. Однако выполняемые ею проверки не отслеживают все виды неверного ввода. Например, предполагается, что отрицательная денежная сумма вводится в формате -$9.95, но если пользователь по ошибке введет ее как $-9.95, функция не выведет сообщения об ошибке и объекту Money будет присвоено неправильное значение. Какое значение прочитает функция input в этом случае? Какую дополнительную проверку вы бы добавили, чтобы выявить большинство ошибок, вызванных неправильным расположением знака минус?

7. В разделе «Ловушка: ведущие нули в числовых константах» предлагается написать короткую программу, проверяющую, интерпретирует ли ваш компилятор ведущий нуль как указание на то, что вводится число в восьмеричной системе. Напишите эту программу.

Квалификатор const

Передача параметров по ссылке более эффективна, чем передача по значению. Полученный по значению параметр является локальной переменной, инициализированной значением аргумента, так что во время работы функции в памяти находятся две копии аргумента. Параметр, передаваемый по ссылке, непосредственно заменяется аргументом, и во время работы функции в памяти находится только одна копия аргумента. Для параметров простых типов, таких как int или double, различие в эффективности незначительно, но для параметров типа класса оно может быть существенным. Поэтому имеет смысл передавать параметры типа класса по ссылке, а не по значению, даже если функция их не изменяет.

Когда параметр передается по ссылке и функция не меняет его значение, этот параметр можно пометить для компилятора как неизменяемый. При этом перед именем типа параметра помещают квалификатор const, и такой параметр называют константным. В качестве примера еще раз рассмотрим класс Money, объявленный в программе листинга 8.3. Параметры-объекты дружественной функции add этого класса можно сделать константными следующим образом:

```
class Money
{
public:
    friend Money add(const Money& amount1, const Money& amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает сумму значений
    // аргументов amount1 и amount2.
        ...
```

В случае использования константного параметра квалификатор const должен быть включен и в объявление, и в заголовок определения функции. Поэтому приведенная выше модификация определения класса Money требует следующего изменения заголовка определения функции add:

```
Money add(const Money& amount1, const Money& amount2)
{
    ...
```

Остальная часть определения функции будет такой же, как в листинге 8.3.

Квалификатор const в объявлении параметров предназначен для автоматического контроля ошибок. Если определение функции содержит ошибку, заключающуюся в случайном изменении константного параметра, то будет выведено соответствующее сообщение. Данный квалификатор может использоваться с параметрами любых типов; однако чаще всего он применяется для параметров типа класса, передаваемых по ссылке (и иногда для других параметров, значения которых очень велики).

Параметры, передаваемые по ссылке, заменяются аргументами при вызове функции, и эта функция может изменять значения аргументов. При вызове функции-члена вызывающий объект во многом подобен параметру, переданному по ссылке, –

ведь функция может изменять его значение. Рассмотрим сказанное на примере следующего фрагмента кода (класс Money определен в листинге 8.3):

```
Money m;
m.input(cin);
```

При объявлении объекта m переменная-член all_cents инициализируется значением 0. Вызов функции-члена input заменяет его значением, введенным пользователем. Таким образом, вызов m.input(cin) изменяет значение объекта m, как если бы он был аргументом, передаваемым по ссылке.

Квалификатор const применяется к вызывающим объектам так же, как к параметрам. Когда класс содержит функцию-член, которая не должна изменять значение вызывающего объекта, можно пометить ее с помощью квалификатора const. Тогда, если функция случайно изменит значение вызывающего объекта, будет отображено сообщение об ошибке. В случае использования функции-члена квалификатор const ставится в конце ее объявления перед точкой с запятой:

```
class Money
{
    public:
        ...
    void output(ostream& outs) const;
        ...
```

Квалификатор const должен быть задан и в объявлении, и в определении функции, то есть определение функции output должно начинаться так:

```
void Money::output(ostream& outs) const
{
    ...
```

Остальная часть определения функции будет такой же, как в листинге 8.3.

Ловушка: непоследовательное использование квалификатора const

Квалификатор const следует использовать по принципу «все или ничего». Если он задан для одного параметра некоторого типа, то должен быть задан и для всех остальных параметров того же типа, не изменяемых вызовом функции. Более того, если это тип класса, квалификатор const следует задать для каждой функции-члена, которая не меняет значение вызывающего объекта. Данное требование связано с возможностью вызова одних функций из других. В качестве примера рассмотрим следующее определение функции guarantee:

```
void guarantee(const Money& price)
{
    cout << "If not satisfied, we will pay you\n"
         << "double your money back.\n"
         << "That's a refund of $"
         << (2*price.get_value()) << endl;
}
```

Если не включить квалификатор в определение функции-члена get_value, большинство компиляторов выведет для функции guarantee сообщение об ошибке.

Функция-член get_value не изменяет вызывающий объект price, но когда компилятор обрабатывает определение функции guarantee, он полагает, что функция get_value изменяет (или, по крайней мере, может изменить) значение этого объекта. Дело в том, что он транслирует определение функции guarantee, и в это время ему ничего не известно о функции-члене get_value, кроме ее объявления. Когда таковое не содержит квалификатора const, указывающего компилятору, что вызывающий объект не будет изменен, компилятор считает такое изменение возможным, поэтому не «заглядывает» в определение функции get_value, чтобы узнать, так ли это на самом деле. По этой причине, если для параметра типа Money используется квалификатор const, все функции-члены класса Money, не изменяющие вызывающий объект, следует объявить с данным квалификатором, в частности, он должен присутствовать в объявлении функции get_value.

Квалификатор const параметра

Если поместить квалификатор const перед именем типа параметра, передаваемого по ссылке, параметр станет константным. (Квалификатор const должен быть задан и в объявлении, и в заголовке определения функции.) Этот квалификатор указывает компилятору, что данный параметр не должен изменяться функцией. Когда по каким-либо причинам определение функции содержит операторы, изменяющие значение такого параметра, компилятор выдает сообщение об ошибке. Параметры типа класса, не изменяемые функцией, лучше объявлять константными и передавать по ссылке, а не по значению.

Если функция-член не меняет значение вызывающего объекта, эту функцию можно пометить как константную, добавив в ее объявление квалификатор const. Если же в константной функции случайно окажутся операторы, изменяющие вызывающий объект, компилятор выдаст сообщение об ошибке. Квалификатор const добавляется в конец объявления функции-члена перед точкой с запятой. В заголовок определения этой функции-члена тоже нужно добавить квалификатор const, чтобы заголовок соответствовал ее объявлению.

Пример

```
class Sample
{
public:
    Sample();
    friend int compare(const Sample& s1, const Sample& s2);
    void input();
    void output() const;
private:
    int stuff;
    double more_stuff;
};
```

Используйте квалификатор const по принципу «все или ничего». Он должен либо задаваться для всех неизменяемых параметров типа класса и для всех функций-членов класса, не изменяющих вызывающий объект, либо вовсе не использоваться.

В листинге 8.4 приведено модифицированное определение класса Money из листинга 8.3, содержащее квалификаторы const везде, где они уместны. Определения

функций-членов и дружественных функций остались прежними с той разницей, что в заголовки многих из них добавлен квалификатор const, чтобы заголовки соответствовали объявлениям функций, приведенным в листинге 8.4.

Листинг 8.4. Класс Money с константными параметрами

```
// Класс для денежных сумм в валюте США.
class Money
{
public:
    friend Money add(const Money& amount1, const Money& amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает сумму значений аргументов amount1 и amount2.

    friend bool equal(const Money& amount1, const Money& amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает true, если аргументы amount1 и amount2 имеют одинаковые
    // значения; в противном случае возвращает false.

    Money(long dollars, int cents);
    // Инициализирует объект таким образом, чтобы он
    // представлял заданное в аргументах количество
    // долларов и центов. Если значение денежной суммы отрицательное,
    // значения аргументов dollars и cents должны быть отрицательными.

    Money(long dollars);
    // Инициализирует объект таким образом,
    // чтобы он представлял значение $dollars.00.

    Money();
    // Инициализирует объект таким образом,
    // чтобы он представлял значение $0.00.

    double get_value() const;
    // Предусловие: вызывающему объекту присвоено значение.
    // Возвращает значение денежной суммы, записанное в вызывающем объекте.

    void input(istream& ins);
    // Предусловие: если ins – входной файловый поток, то он
    // уже соединен с файлом. В этот поток введено значение денежной суммы,
    // включая символ доллара. Отрицательные значение денежных сумм вводятся
    // как -$100.00.
    // Постусловие: вызывающему объекту присвоено значение денежной суммы,
    // прочитанное из входного потока ins.

    void output(ostream& outs) const;
    // Предусловие: если outs – файловый выходной поток,
    // то он уже соединен с файлом.
    // Постусловие: символ доллара и значение денежной суммы, хранящееся
    // в вызывающем объекте, записаны в выходной поток outs.
private:
    long all_cents;
};
```

Упражнения для самопроверки

8. Приведите полное определение функции-члена get_value для определения класса Money из листинга 8.4.
9. Почему такое добавление квалификатора const:

```
class Money
{
        ...
public:
    void input(istream& ins) const;
        ...
```

в объявление функции-члена input класса Money, приведенное в листинге 8.4, будет неправильным?

10. В чем различие и сходство параметра, передаваемого по значению и константного параметра, передаваемого по ссылке? Отвечая на этот вопрос, воспользуйтесь примерами:

```
void call_by_value(int x);
void call_by_const_reference(const int & x);
```

11. Рассмотрите следующие определения:

```
const int x = 17;
class A
{
public:
    A();
    A(int x);
    int f()const;
    int g(const A& x);
private:
    int i;
};
```

Каждое из трех ключевых слов const указывает компилятору на требование, которое он должен соблюсти. Что это за требование в каждом из трех случаев?

8.2. Перегрузка операторов

Он безупречный оператор.

Строка из песни

Выше было рассказано, как написать дружественную функцию add для класса Money и использовать ее, чтобы произвести сложение двух объектов типа Money (см. листинг 8.3). Эта функция успешно выполняет операцию сложения значений объектов, но гораздо проще воспользоваться оператором +, как в следующем фрагменте кода:

```
Money total, cost, tax;
cout << "Enter cost and tax: ";
cost.input(cin);
```

```
tax.input(cin);
total = cost + tax;
```

вместо того, чтобы выполнять достаточно громоздкий вызов

```
total = add(cost, tax);
```

Вспомните, что операторы, выполняющие различные арифметические и логические операции, подобны функциям, и отличаются от них только используемым синтаксисом. Если в вызове обычной функции аргументы задаются в скобках после ее имени, вот так:

```
add(cost, tax)
```

то для бинарного оператора аргументы задаются с двух сторон от знака операции:

```
cost + tax
```

Функция может быть перегружена, для того чтобы принимать аргументы разных типов. Поскольку оператор – та же функция, его тоже можно перегружать. Причем способ его перегрузки практически не отличается от способа перегрузки имени обычной функции. В этом разделе мы покажем, как перегружать операторы C++.

Реализация перегрузки операторов

Оператор + и многие другие операторы языка C++ можно перегружать, чтобы они принимали аргументы типа класса. Разница между перегрузкой оператора + и определением функции add (см. листинг 8.3) заключается лишь в небольшом изменении синтаксиса. Определения перегруженного оператора и функции add отличаются только тем, что вместо имени add используется имя +, предваряемое ключевым словом operator. В листинге 8.5 приведена небольшая демонстрационная программа, включающая еще одну версию типа Money с определением перегруженного оператора +.

Листинг 8.5. Перегрузка операторов

```
// Программа, в которой используется класс Money. (Это улучшенная версия
// класса Money, приведенного в листинге 8.3 и модифицированного в листинге 8.4.)
#include <iostream>
#include <cstdlib>
#include <cctype>
using namespace std;

// Класс для денежных сумм в валюте США.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает сумму значений аргументов amount1 и amount2.

    friend bool operator ==(const Money& amount1, const Money& amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает true, если аргументы amount1 и amount2 содержат одинаковые
    // значения; в противном случае возвращает false.
```

```
    Money(long dollars, int cents);

    Money(long dollars);

    Money();

    double get_value() const;

    void input(istream& ins);

    void output(ostream& outs) const;
private:
    long all_cents;
/* Некоторые комментарии листинга 8.4 удалены в целях экономии
   места, но в реальной программе они нужны. */
};
```

... Здесь располагаются дополнительные определения функций, содержавшиеся в листинге 8.3. ...

```
int main()
{
    Money cost(1, 50), tax(0, 15), total;
    total = cost + tax;

    cout << "cost = ";
    cost.output(cout);
    cout << endl;
    cout << "tax = ";
    tax.output(cout);
    cout << endl;
    cout << "total bill = ";
    total.output(cout);
    cout << endl;
    if (cost == tax)
        cout << "Move to another state.\n";
    else
        cout << "Things seem normal.\n";
    return 0;
}

Money operator +(const Money& amount1, const Money& amount2)
{
    Money temp;
    temp.all_cents = amount1.all_cents + amount2.all_cents;
    return temp;
}

bool operator == (const Money& amount1, const Money& amount2)
{
    return (amount1.all_cents == amount2.all_cents);
}
```

Вывод на экран

```
cost = $1.50
tax = $0.15
total bill = $1.65
Things seem normal.
```

Определения функций-членов такие же, как в листинге 8.3, с той разницей, что в заголовки функций добавлены квалификаторы const для их соответствия объявлениям функций в приведенном выше определении класса Money. Никакие другие изменения в определениях функций-членов не нужны.

В этом листинге класс Money содержит определение еще одного перегруженного оператора, ==, используемого для сравнения двух объектов типа Money. Как показано в листинге, если amount1 и amount2 — два объекта типа Money, выражение

```
amount1 == amount2
```

возвращает то же, что и логическое выражение

```
amount1.all_cents == amount2.all_cents
```

Перегрузку допускает большинство, но не все операторы C++. Перегружаемый оператор не обязательно должен быть дружественным к классу, но как правило, это создает определенные удобства. Ниже во врезке «Правила перегрузки операторов» приведены некоторые технические подробности того, как и когда можно выполнять перегрузку.

Перегрузка операторов

Бинарный оператор, такой как +, -, /, % и т. д. — это просто функция с особым синтаксисом вызова. В вызове функции-оператора аргументы задаются до и после знака операции, а для обычной функции они указываются в круглых скобках после ее имени. Определение оператора похоже на определение обычной функции с той разницей, что вместо имени функции в его заголовке стоит ключевое слово operator, за которым следует знак определяемой операции (имя). Предопределенные операторы, такие как + и ему подобные, можно перегружать для поддержки операций над типами классов.

Оператор может быть дружественным к классу, но это не обязательно. Пример перегрузки оператора + в виде дружественной функции приведен в листинге 8.5.

Упражнения для самопроверки

12. Каково различие между бинарным оператором и функцией?

13. Предположим, вы хотите перегрузить оператор < так, чтобы с его помощью можно было сравнивать объекты типа Money, определенного в листинге 8.5. Что для этого нужно добавить в определение класса Money?

14. Допустим, вы хотите перегрузить оператор <= таким образом, чтобы его можно было применять к объектам типа Money, определенного в листинге 8.5. Что для этого нужно добавить в определение класса Money?

15. Можно ли с помощью перегрузки операторов изменить поведение оператора + для целых чисел? Поясните свой ответ.

Правила перегрузки операторов

- При перегрузке оператора как минимум один из аргументов его новой версии должен иметь тип класса.
- Перегруженный оператор может, но не обязательно должен, быть дружественным к классу; функция-оператор может быть как членом класса, так и обычной функцией.
- Создать новые операторы невозможно, допустима только перегрузка существующих, таких как +, -, *, /, % и т. п.
- Нельзя изменить количество аргументов оператора, например, путем перегрузки превратить оператор % из бинарного в унарный или оператор ++ из унарного в бинарный.
- Нельзя изменять приоритет оператора. Перегруженный оператор имеет тот же приоритет, что и его исходная версия. Так, `x*y + z` всегда означает `(x*y) + z`, даже если `x`, `y` и `z` являются объектами и операторы + и * перегружены для их класса.
- Запрещена перегрузка следующих операторов: оператора точка (`.`), оператора доступа к члену класса по имени класса (`::`) и операторов `.*` и `?:`, не описанных в этой книге.
- Оператор присваивания может быть перегружен так, что его значение будет заменено новым, но это делается иным способом. О перегрузке оператора = рассказывается в главе 12. Некоторые другие операторы, включая `[]` и `->`, тоже перегружаются другим способом. Об этих операторах рассказывается далее.

Конструкторы для автоматического приведения типов

Если определение класса содержит необходимые конструкторы, система автоматически выполняет приведение типа. Например, когда в программе находится определение класса `Money`, приведенное в листинге 8.5, в ней может выполняться такой код:

```
Money base_amount(100, 60), full_amount;
full_amount = base_amount + 25;
full_amount.output(cout);
```

Он выведет следующее:

```
$125.60
```

Этот код выглядит просто и естественно, но содержит интересную деталь — число 25 в выражении `base_amount + 25` не является объектом класса `Money`. В листинге 8.5 оператор + перегружен таким образом, что принимает два объекта типа `Money`. Мы не включали в программу его перегруженную версию для сложения объекта типа `Money` и целого числа. А константа 25 может рассматриваться как значение типа `int` или `long`, но никак не `Money`, если только в определении класса не сказано, как преобразовать целое число в объект типа `Money`. Единственный способ, который позволит системе трактовать число 25 как $25,00, — это включение в класс конструктора, принимающего аргумент типа `long`. Встретив выражение

```
base_amount + 25
```

система вначале проверяет, имеется ли перегруженная версия оператора +, для сложения значения типа Money и целого числа. Поскольку такой версии нет, система посмотрит, нет ли у класса конструктора, принимающего единственный аргумент типа long. Если такой конструктор имеется, с его помощью число 25 будет преобразовано в значение типа Money. Конструктор с аргументом типа long «говорит» системе, как это сделать. В его определении сказано, что, создав новый пустой объект типа Money, нужно присвоить его переменной-члену all_ cents значение 2500. Так конструктор преобразует число 25 в $25,00. (Определение этого конструктора приведено в листинге 8.3.)

Обратите внимание, что данное преобразование типа не работает без подходящего конструктора. Например, у типа Money (см. листинг 8.5) нет конструктора, принимающего аргумент типа double, поэтому оператор

```
full_amount = base_amount + 25.67;
```

недопустим, и при его компиляции в составе программы будет выведено сообщение об ошибке. Для того чтобы он стал допустимым, нужно изменить определение класса Money, добавив еще один конструктор со следующим объявлением:

```
class Money
{
public:
    ...
    Money(double amount);
    // Инициализирует объект таким образом,
    // чтобы он представлял значение $amount.
    ...
```

Определение этого конструктора вам предлагается написать в упражнении 16.

Описанное автоматическое преобразование типа, выполняемое с помощью конструктора, наиболее типично для перегруженных арифметических операторов, например + и -. Однако оно применимо и к аргументам обычных функций, функций-членов и других перегруженных операторов.

Упражнение для самопроверки

16. Приведите определение конструктора, описанного в конце предыдущего раздела. Этот конструктор предназначен для класса Money, приведенного в листинге 8.5. Его определение начинается так:

```
Money::Money(double amount)
{
```

Перегрузка унарных операторов

Помимо бинарных операторов, таких как оператор +, используемый в выражениях типа x + y, в C++ есть унарные операторы, например оператор -, меняющий знак числового операнда на противоположный. Так, оператор

```
x = -y;
```

присваивает переменной x значение, равное значению переменной y, взятому с противоположным знаком.

Другими примерами унарных операторов являются операторы инкрементирования ++, и декрементирования --.

Унарные операторы перегружаются так же, как бинарные. Скажем, можно переопределить класс Money, приведенный в листинге 8.5, чтобы для него поддерживались и унарный и бинарный операторы -. Это новое определение класса Money приведено в листинге 8.6. Предположим, ваша программа содержит определение класса и следующий код:

```
Money amount1(10), amount2(6), amount3;
```

Тогда оператор

```
amount3 = amount1 - amount2;
```

присваивает объекту amount3 значение, которое равно разности значений объектов amount1 и amount2. Затем оператор

```
amount3.output(cout);
```

выведет на экран значение $4.00.

Следующий оператор:

```
amount3 = -amount1;
```

присваивает объекту amount3 значение, равное значению объекта amount1, взятому с противоположным знаком. После этого оператор:

```
amount3.output(cout);
```

выведет на экран значение -$10.00.

Для перегрузки операторов ++ и -- можно взять за образец код перегрузки унарного оператора -, приведенный в листинге 8.6. Перегруженное определение будет применяться к оператору при его использовании в префиксной позиции, например, ++x и --x. Постфиксные версии этих операторов обрабатываются иным способом, который в этой книге не описан.

Листинг 8.6. Перегрузка унарного оператора

```
// Класс для значений денежных сумм в валюте США.
class Money   // Это дополненная версия класса Money, приведенного в листинге 8.5.
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);

    friend Money operator -(const Money& amount1, const Money& amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает разность значений аргументов amount1 и amount2.

    friend Money operator -(const Money& amount);
    // Предусловие: аргумент amount содержит значение.
    // Возвращает значение аргумента amount,
    // взятое с противоположным знаком.
```

продолжение

Листинг 8.6 (*продолжение*)

```
    friend bool operator ==(const Money& amount1, const Money& amount2);

/* Мы опустили директивы include и некотрые комментарии,
   но в реальной программе они необходимы. */
    Money(long dollars, int cents);

    Money(long dollars);

    Money();

    double get_value() const;

    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long all_cents;
};

... Здесь располагаются определения других функций и функция main. ...

Money operator -(const Money& amount1, const Money& amount2)
{
    Money temp;
    temp.all_cents = amount1.all_cents - amount2.all_cents;
    return temp;
}

Money operator -(const Money& amount)
{
    Money temp;
    temp.all_cents = -amount.all_cents;
    return temp;
}
```

Определения остальных функций такие же, как в листинге 8.5.

Перегрузка операторов >> и <<

Оператор вывода, использующийся с потоком cout, — это бинарный оператор, подобный операторам + и -. Рассмотрим следующую строку кода:

```
cout << "Hello out there.\n";
```

Здесь << является оператором, cout — первым операндом, а строковое значение "Hello out there.\n" — вторым операндом. Любой из операндов можно изменить. Например, если переменная fout представляет выходной поток типа ofstream, подключенный к файлу с помощью функции open, то в приведенном выше операторе можно заменить cout операндом fout, и заданная в нем строка будет выведена не на экран, а в файл. Строку "Hello out there.\n", разумеется, тоже легко заменить другой строкой, переменной или числом. Оператор вывода << можно перегружать,

подобно другим бинарным операторам, но эта задача несколько более сложная, чем перегрузка арифметических операторов.

Во всех наших определениях класса Money (см. листинги 8.3–8.6) для вывода значений типа Money на экран использовалась функция-член output. Она справлялась со своей задачей, но гораздо удобнее было бы пользоваться привычным оператором <<, скажем так:

```
Money amount(100);
cout << "I have " << amount << " in my purse.\n";
```

вместо того чтобы вызывать функцию output следующим образом:

```
Money amount(100);
cout << "I have ";
amount.output(cout);
cout << " in my purse.\n";
```

Прежде чем выполнять перегрузку оператора <<, важно выяснить, что он должен вернуть при использовании в выражениях, подобных приведенному ниже:

```
cout << amount
```

Двумя операндами здесь являются cout и amount, и в результате вычисления выражения на экран должно быть выведено значение объекта amount. Если << — такой же оператор, как + или *, это выражение должно возвращать какое-нибудь значение, поскольку выражения с любыми другими операторами типа n1 + n2 возвращают значения. Что же должно вернуть выражение cout << amount? Чтобы ответить на это вопрос, нужно рассмотреть более сложное выражение с оператором <<, например следующее, состоящее из цепочки выражений с названным оператором:

```
cout << "I have " << amount << " in my purse.\n";
```

Если оператор << аналогичен другим операторам (таким, как +), приведенное выражение должно быть (и на самом деле является) эквивалентным следующему:

```
((cout << "I have ") << amount) << " in my purse.\n";
```

Теперь выясним, что же должен возвращать оператор <<, чтобы это выражение имело смысл.

Сначала вычисляется выражение

```
(cout << "I have")
```

Чтобы вся конструкция работала, оно должно возвращать объект cout, и тогда вычисления можно будет продолжить так:

```
(cout << amount) << " in my purse.\n";
```

Очевидно, что для выполнения оставшегося выражения выражение (cout << amount) тоже должно вернуть cout.

```
cout << " in my purse.\n";
```

Рассмотренный нами процесс показан на рис. 8.1. Оператор << должен возвращать свой первый аргумент, которым является поток типа ostream.

```
cout << "I have " << amount << " in my purse.\n";
```

означает то же самое, что

```
((cout << "I have ") << amount) << " in my purse.\n";
```

и вычисляется следующим образом.

Сначала вычисляется выражение (cout << "I have "), возвращающее cout:

```
((cout << "I have ") << amount) << " in my purse.\n";
```

и выводится строка "I have".

```
(cout << amount) << " in my purse.\n";
```

Затем вычисляется выражение (cout << amount), возвращающее cout:

```
(cout << amount) << " in my purse.\n";
```

и выводится значение объекта amount.

```
cout << " in my purse.\n";
```

Потом вычисляется выражение cout << " in my purse.\n", также возвращающее cout:

```
cout << " in my purse.\n";
```

и выводится строка " in my purse.\n".

```
cout;
```

Поскольку операторов << больше не осталось, процесс завершается.

Рис. 8.1. Использование << в качестве оператора

Итак, объявление перегруженного оператора для использования с классом Money может быть таким:

```
class Money
{
public:
    ...
    friend ostream& operator <<(ostream& outs, const Money& amount);
    // Предусловие: если outs – файловый выходной поток,
    // то он уже соединен с файлом.
    // Постусловие: символ доллара и значение денежной суммы, хранящееся
    // в вызывающем объекте, записаны в выходной поток outs.
    ...
```

После перегрузки оператора вывода << нам больше не нужна функция-член output, поэтому можно удалить ее из определения класса Money. Определение перегруженного оператора << очень похоже на определение функции-члена output. Вот что оно собой представляет:

```
ostream& operator <<(ostream& outs, const Money& amount)
{
```

```
    ... Эта часть в точности такая же, как тело
    функции Money::output, приведенное в листинге 8.3
    (только all_cents заменено amount.all_cents). ...

    return outs;
}
```

В приведенном объявлении и определении перегруженного оператора << осталось объяснить только одно — назначение символа & в определении возвращаемого типа ostream&. Проще всего сделать это так: когда оператор или функция возвращает поток, в конец имени типа возвращаемого значения следует добавлять символ &. Данное простое правило позволяет перегружать операторы << и >>, однако пока мы лишь сформулировали правило, теперь же объясним, что означает символ &. Добавляя к имени возвращаемого типа этот символ, вы указываете компилятору, что данный оператор или функция возвращает ссылку. Все встречавшиеся нам до сих пор операторы и функции возвращали значения. Но если возвращаемым типом является поток, нельзя просто вернуть значение потока. Его значением является «весь файл или клавиатура, или экран», и трудно представить, какой смысл можно вложить в «возврат» подобных вещей. Таким образом, нам нужно вернуть сам поток, а не его значение. Добавляя символ & к имени возвращаемого типа, вы указываете, что оператор или функция возвращает ссылку, то есть сам объект, а не его значение.

Оператор >> перегружается аналогично оператору <<, но вторым его аргументом является объект, получающий входное значение. Поэтому вторым параметром оператора должен быть обычный параметр, передаваемый по ссылке. Вот как определяется перегруженный оператор >>:

```
istream& operator >>(istream& ins, Money& amount)
{
    ... Эта часть в точности такая же, как тело
    функции Money::output, приведенное в листинге 8.3
    (только all_cents заменено amount.all_cents). ...

    return ins;
}
```

Полное определение перегруженных операторов ввода и вывода приведено в листинге 8.7, где показана еще одна версия класса Money. На этот раз функции output и input заменены в нем перегруженными операторами >> и <<, которые теперь можно применять для ввода и вывода значений типа Money.

Перегрузка операторов >> и <<

Операторы ввода и вывода можно перегружать точно так же, как любые другие операторы C++. Каждый из этих операторов должен возвращать поток. В конец имени типа возвращаемого перегруженным оператором значения нужно добавить символ &. Ниже приведены объявления и начало определений обеих функций. Пример содержится в программе из листинга 8.7.

продолжение

Объявления функций

```
class имя_класса
{
public:
        ...
    friend istream& operator >>(istream& параметр_1,
                                имя_класса& параметр_2);
    // Параметр_1 – параметр для потока.
    friend ostream& operator <<(ostream& параметр_3,
                                const имя_класса& парметр_4);
    // Параметр_3 – параметр для объекта, принимающего ввод.
        ...
```

Определения

```
istream& operator >>(istream& параметр_1,
                                имя_класса& параметр_2)
{
...
}
ostream& operator <<(ostream& параметр_3,
                             const имя_класса& параметр_4)
{
...
}
```

Листинг 8.7. Перегрузка операторов >> и <<

```
/* Это более полная версия кода класса Money, приведенного в листинге 8.6

   Мы опустили комментарии, содержащиеся в листингах 8.5 и 8.6,
   но в реальной программе они необходимы. */

// Программа, в которой используется класс Money.
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cctype>
using namespace std;

// Класс, определяющий денежные суммы в валюте США.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);

    friend Money operator -(const Money& amount1, const Money& amount2);

    friend Money operator -(const Money& amount);

    friend bool operator ==(const Money& amount1, const Money& amount2);

    Money(long dollars, int cents);

    Money(long dollars);
```

```
    Money();

    double get_value() const;

    friend istream& operator >>(istream& ins, Money& amount);
    // Перегружает оператор >>, чтобы им можно было пользоваться
    // для ввода значений типа Money.
    // Отрицательные значения денежных сумм вводятся как -$100.00.
    // Предусловие: если ins – файловый входной поток,
    // то он уже соединен с файлом.

    friend ostream& operator <<(ostream& outs, const Money& amount);
    // Перегружает оператор <<, чтобы им можно было пользоваться
    // для вывода значений типа Money.
    // Перед каждым значением типа Money выводит символ доллара.
    // Предусловие: если outs – файловый выходной поток,
    // то он уже соединен с файлом.
private:
    long all_cents;
};

int digit_to_int(char c);
// Используется в определении перегруженного оператора ввода >>.
// Предусловие: c – одна из цифр от '0' до '9'.
// Возвращает целочисленное значение символа, представляющего данную цифру;
// например, digit_to_int('3') возвращает 3.

int main()
{
    Money amount;
    ifstream in_stream;
    ofstream out_stream;

    in_stream.open("infile.dat");
    if (in_stream.fail())
    {
        cout << "Input file opening failed.\n";
        exit(1);
    }

    out_stream.open("outfile.dat");
    if (out_stream.fail())
    {
        cout << "Output file opening failed.\n";
        exit(1);
    }

    in_stream >> amount;
    out_stream << amount
               << " copied from the file infile.dat.\n";
    cout << amount
         << " copied from the file infile.dat.\n";

    in_stream.close();
    out_stream.close();

    return 0;
}
```

продолжение

Листинг 8.7 (*продолжение*)

```
// Используем библиотеки классов iostream, cctype, cstdlib.
istream& operator >>(istream& ins, Money& amount)
{
    char one_char, decimal_point,
         digit1, digit2;   // Цифры, определяющие количество центов.
    long dollars;
    int cents;
    bool negative;   // Устанавливается в true, если введенное
                     // значение суммы отрицательно.

    ins >> one_char;
    if (one_char == '-')
    {
        negative = true;
        ins >> one_char;   // Считывает '$'.
    }
    else
        negative = false;
    // Если ввод допустим, то one_char == '$'.

    ins >> dollars >> decimal_point >> digit1 >> digit2;

    if (one_char != '$' || decimal_point != '.'
         || !isdigit(digit1) || !isdigit(digit2))
    {
        cout << "Error illegal form for money input\n";
        exit(1);
    }

    cents = digit_to_int(digit1)*10 + digit_to_int(digit2);

    amount.all_cents = dollars*100 + cents;
    if (negative)
        amount.all_cents = -amount.all_cents;

return ins;
}

int digit_to_int(char c)
{
    return (int(c) - int('0'));
}

// Используем библиотеки классов cstdlib и iostream.
ostream& operator <<(ostream& outs, const Money& amount)
{
    long positive_cents, dollars, cents;
    positive_cents = labs(amount.all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (amount.all_cents < 0)
        outs << "-$" << dollars << '.';
    else
        outs << "$" << dollars << '.';

    if (cents < 10)
```

```
        outs << '0';
    outs << cents;
return outs;
}
```

infile.dat
(Не изменяется программой)

```
      $1.11 $2.22
  $3.33
```

outfile.dat
(После выполнения программы)

```
$1.11 copied from the file
infile.dat.
```

Вывод на экран

```
$1.11 copied from the file infile.dat.
```

Определения остальных функций-членов и перегруженных операторов в приведенном листинге такие же, как в листингах 8.3. 8.4, 8.5 и 8.6.

Упражнения для самопроверки

17. Вот определение класса Pairs:

```
#include <iostream>
using namespace std;
class Pairs
{
public:
    Pairs();
    Pairs(int first, int second);
    // Другие члены и дружественные функции.
    friend istream& operator>> (istream& ins,
                                Pairs& second);
    friend ostream& operator<< (ostream& outs,
                                const Pairs& second);
private:
    int f;
    int s;
};
```

Объекты типа Pairs могут использоваться, когда в программе нужны упорядоченные пары значений. Ваша задача — написать реализации перегруженных операторов ввода и вывода, чтобы значения объектов этого класса можно было вводить и выводить в форме (5,6), (5,-4), (-5,4) или (-5,-6). Ни конструкторы, ни другие члены класса писать не нужно, как нет необходимости проверять формат вводимых данных.

18. Ниже приведено определение класса Percent.

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

class Percent
{
public:
    friend bool operator ==(const Percent& first,
                            const Percent& second);
    friend bool operator <(const Percent& first,
                           const Percent& second);
    Percent();
    Percent(int percent_value);
```

```
    friend istream& operator >>(istream& ins,
                                Percent& the_object);
    // Перегружает оператор >>, чтобы им можно было пользоваться
    // для ввода значений типа Percent.
    // Предусловие: если ins – файловый входной поток,
    // то он уже соединен с файлом.

    friend ostream& operator <<(ostream& outs,
                                const Percent& a_percent);
    // Перегружает оператор <<, чтобы им можно было пользоваться
    // для вывода значений типа Percent.
    // Предусловие: если outs – файловый выходной поток,
    // то он уже соединен с файлом.
private:
    int value;
};
```

Объекты типа Percent представляют количество процентов, скажем 10 % или 99 %. Приведите определение перегруженных операторов >> и << для ввода и вывода значений объектов этого класса. Предполагается, что входное значение всегда представляет собой целое число, за которым следует символ % (например, 25 %). Значения объектов класса являются целыми числами и хранятся в переменной-члене value типа int. Ни конструкторы, ни другие перегруженные операторы реализовывать не нужно – только операторы ввода и вывода.

Резюме

- Дружественная функция класса – это обычная функция, имеющая доступ к закрытым членам класса подобно его функциям-членам.
- Если класс содержит полный набор аксессоров и мутаторов, единственной причиной, по которой функция определяется как дружественная, является желание упростить и сделать более эффективным ее определение.
- Параметр типа класса, не изменяемый функцией, обычно лучше определить как константный.
- Большинство операторов C++ (таких, как + и ==) можно перегружать для использования с объектами определяемых программистом классов.
- При перегрузке операторов ввода и вывода возвращаемым типом данных должен быть поток; причем в конец имени его типа следует добавлять символ &, поскольку необходимо, чтобы он возвращался по ссылке.

Ответы к упражнениям для самопроверки

1.
```
bool before(DayOfYear date1, DayOfYear date2)
{
    return ((date1.get_month() < date2.get_month())
          || (date1.get_month() == date2.get_month()
                && date1.get_day() < date2.get_day()));
}
```

Возвращаемое функцией логическое выражение означает, что дата date1 предшествует дате date2, если месяц даты date1 наступает раньше месяца даты date2 либо их месяцы одинаковы, а день даты date1 наступает раньше дня даты date2.

2. Дружественная функция и функция-член класса сходны тем, что им обеим доступны все члены класса, как открытые, так и закрытые. Однако дружественная функция определяется и используется как обычная функция – в ее вызове не требуется оператор точка, а в определении не нужен спецификатор типа. А функция-член вызывается с использованием имени объекта и оператора точка. Кроме того, определение функции-члена содержит спецификатор типа, включающий имя класса и оператор ::.

3. Ниже приведено модифицированное определение класса DayOfYear. Новая часть кода выделена. Для экономии места некоторые комментарии опущены, но в реальной программе все комментарии, приведенные в листинге 8.2, должны присутствовать.

```
class DayOfYear
{
public:
    friend bool equal(DayOfYear date1, DayOfYear date2);

    friend bool after(DayOfYear date1, DayOfYear date2);
    // Предусловие: аргументы date1 и date2 содержат значения.
    // Возвращает true, если дата дата date1 приходится позже даты date2;
    // в противном случае возвращает false.

    DayOfYear(int the_month, int the_day);
    DayOfYear();
    void input();
    void output();
    int get_month();
    int get_day();
private:
    void check_date();
    int month;
    int day;
};
```

Кроме того, нужно добавить следующее определение функции after:

```
bool after(DayOfYear date1, DayOfYear date2)
{
    return ((date1.month > date2.month) ||
((date1.month == date2.month)
                && (date1.day > date2.day)));
}
```

4. Ниже приведено модифицированное определение класса Money. Новая часть кода выделена. Для экономии места некоторые комментарии опущены, но в реальной программе все комментарии, приведенные в листинге 8.3, должны присутствовать.

```
class Money
{
public:
    friend Money add(Money amount1, Money amount2);

    friend Money subtract(Money amount1, Money amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает разность значений amount1 и amount2.

    friend bool equal(Money amount1, Money amount2);
    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double get_value();
    void input(istream& ins);
    void output(ostream& outs);
private:
    long all_cents;
};
```

Кроме того, нужно добавить следующее определение функции subtract:

```
Money subtract(Money amount1, Money amount2)
{
    Money temp;
    temp.all_cents = amount1.all_cents - amount2.all_cents;
    return temp;
}
```

5. Ниже приведено модифицированное определение класса Money. Новая часть кода выделена. Для экономии места некоторые комментарии опущены, но в реальной программе все комментарии, приведенные в листинге 8.3, должны присутствовать.

```
class Money
{
public:
    friend Money add(Money amount1, Money amount2);
    friend bool equal(Money amount1, Money amount2);
    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double get_value();
    void input(istream& ins);

    void output(ostream& outs);
    // Предусловие: если outs – файловый выходной поток,
    // то он уже соединен с файлом.
    // Постусловие: символ доллара и хранящееся в вызывающем
    // объекте значение денежной суммы выведены в поток outs.

    void output();
    // Постусловие: символ доллара и хранящееся в вызывающем
    // объекте значение денежной суммы выведены на экран.
private:
    long all_cents;
};
```

Кроме того, нужно добавить следующее определение функции output, при этом старое определение этой функции тоже остается, так что теперь их два.

```
void Money::output()
{
    output(cout);
}
```

Будет работать и такое более длинное определение функции:

```
// Используем библиотеки классов cstdlib и iostream.
void Money::output()
{
    long positive_cents, dollars, cents;
    positive_cents = labs(all_cents);
    dollars = positive_cents/100;
    cents = positive_cents%100;

    if (all_cents < 0)
        cout << "-$" << dollars << '.';
    else
        cout << "$" << dollars << '.';

    if (cents < 10)
        cout << '0';
    cout << cents;
}
```

Помимо этого, можно перегрузить функцию-член input, чтобы вызов

```
purse.input();
```

означал то же, что и вызов

```
purse.input(cin);
```

И конечно, допустимо объединение этих дополнений с дополнениями из предыдущего упражнения, если создан значительно усовершенствованный класс Money.

6. При введении пользователем $-9.95 вместо -$9.95 функция input прочитает в переменную one_char символ '$', в переменную dollars — значение -9, в переменную decimal_point — символ '.', а в переменные digit1 и digit2 — символы '9' и '5'. В результате переменной-члену dollars будет присвоено число -9, а переменной-члену cents число 95, и значением объекта будет следующее: -$9.00 плюс $0.95, то есть -$8.05. Чтобы программа не допускала такой ошибки, нужно проверить, не является ли значение переменной-члена dollars отрицательным, поскольку при правильно введенном значении в нее должно помещаться абсолютное значение суммы долларов. Для этого нужно переписать проверочную часть функции-члена input следующим образом:

```
if ( one_char != '$' || decimal_point != '.'
        || !isdigit(digit1) || !isdigit(digit2)
        || dollars < 0 )   // Новое условие
{
    cout << "Error illegal form for money input\n";
    exit(1);
}
```

Такая версия функции по-прежнему не будет выводить сообщение об ошибке для неверного значения с нулевым количеством долларов, такого как $-0.95. Но пока вы изучили недостаточно материала, чтобы написать короткую и удобную проверку для этого случая.

7.
```
#include <iostream>
using namespace std;
int main()
{
    int x;
    cin >> x;
    cout << x << endl;
    return 0;
}
```

Если компилятор интерпретирует введенное значение с ведущим нулем как восьмеричное число, то, прочитав число 077, программа выведет 63. Если же компилятор интерпретирует любые входные числа как десятичные независимо от наличия ведущего нуля, программа выведет 77.

8. Единственным изменением по сравнению с версией, приведенной в листинге 8.3, будет добавление в заголовок функции квалификатора const:

```
double Money::get_value() const
{
    return (all_cents * 0.01);
}
```

9. Функция-член input изменяет значение вызывающего объекта, поэтому, если в ее заголовок добавить квалификатор const, компилятор выдаст сообщение об ошибке.

10. Сходство: оба метода защищают аргумент (в вызывающем коде) от изменения. Различие: при передаче по значению создается копия аргумента, поэтому для данного метода требуется больше памяти, чем при передаче по ссылке.

11. В объявлении const int x = 17; ключевое слово const «обещает» компилятору, что программа не изменит значение переменной x.

В объявлении int f()const; ключевое слово const сообщает компилятору, что функция f не изменит данных вызывающего объекта.

В объявлении int g(const A& x); ключевое слово const сообщает компилятору, что функции g не изменит данных объекта, переданного этой функции в качестве аргумента.

12. Различие между бинарным оператором (таким, как +, *, / и т. п.) и функцией заключается в используемом синтаксисе их вызова. В вызове функции аргументы задаются в скобках после ее имени, тогда как аргументы оператора задаются справа и слева от него. Кроме того, в объявлении и определении перегруженного оператора должно быть указано ключевое слово operator.

13. Ниже приведено модифицированное определение класса Money. Новая часть кода выделена. Для экономии места некоторые комментарии опущены, но в реальной программе все комментарии, приведенные в листинге 8.5, должны присутствовать.

```
class Money
{
public:
    friend Money operator +(const Money& amount1,
                            const Money& amount2);
    friend bool operator ==(const Money& amount1,
                            const Money& amount2);

    friend bool operator < (const Money& amount1,
                            const Money& amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает true, если значение аргумента amount1 меньше значения аргумента amount2;
    // в противном случае возвращает false.

    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double get_value() const;
    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long all_cents;
};
```

Кроме того, нужно добавить следующее определение перегруженного оператора <:

```
bool operator < (const Money& amount1,
                 const Money& amount2)
{
    return (amount1.all_cents < amount2.all_cents);
}
```

14. Ниже приведено модифицированное определение класса Money. Новая часть кода выделена. Для экономии места некоторые комментарии опущены, но в реальной программе все комментарии, приведенные в листинге 8.5, должны присутствовать.

```
class Money
{
public:
    friend Money operator +(const Money& amount1,
                            const Money& amount2);
    friend bool operator ==(const Money& amount1,
                            const Money& amount2);

    friend bool operator < (const Money& amount1,
                            const Money& amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает true, если значение аргумента amount1 меньше значения аргумента amount2;
    // в противном случае возвращает false.

    friend bool operator <= (const Money& amount1,
                             const Money& amount2);
    // Предусловие: аргументы amount1 и amount2 содержат значения.
    // Возвращает true, если значение аргумента amount1 меньше или равно
    // значению аргумента amount2; в противном случае возвращает false.
```

```
    Money(long dollars, int cents);
    Money(long dollars);
    Money();
    double get_value() const;
    void input(istream& ins);
    void output(ostream& outs) const;
private:
    long all_cents;
};
```

Кроме того, нужно добавить следующее определение перегруженного оператора <= (а также определение перегруженного оператора <, приведенное в предыдущем упражнении):

```
bool operator <= (const Money& amount1,
                  const Money& amount2)
{
    return ((amount1.all_cents < amount2.all_cents)
          ||(amount1.all_cents == amount2.all_cents));
}
```

15. При перегрузке оператора хотя бы один из его аргументов должен иметь тип класса. Это означает, что для целых чисел оператор сложения перегрузить нельзя, и следовательно, нельзя изменить его поведение. Более того, данное правило фактически запрещает изменение поведения операторов с любыми встроенными типами.

16.
```
// Используем библиотеку классов cmath.
Money::Money(double amount)
{
    all_cents = floor(amount*100);
}
```

Это определение просто игнорирует любые суммы, меньшие одного цента. Например, число 12.34999 оно преобразует в целочисленное значение 1234, представляющее сумму $12,34. Можно определить конструктор, который будет поступать с дробной частью центов иначе.

17.
```
istream& operator>>(istream& ins, Pairs& second)
{
    char ch;
    ins >> ch;   // Удаляем из входного потока скобку '('.
    ins >> second.f;
    ins >> ch;   // Удаляем из входного потока запятую ','.
    ins >> second.s;
    ins >> ch;   // Удаляем из входного потока скобку ')'.
    return ins;
}

ostream& operator<<(ostream& outs, const Pairs& second)
{
    outs << '(';
    outs << second.f;
    outs << ',';   // Можно использовать ", ",
                   // чтобы разделить числа пробелом.
    outs << second.s;
    outs << ')';
```

```
    return outs;
}
```

18.
```
// Используем библиотеку классов iostream.
istream& operator >>(istream& ins, Percent& the_object)
{
    char percent_sign;
    ins >> the_object.value;
    ins >> percent_sign;   // Считывает (удаляя из потока) символ %.
    return ins;
}

// Используем библиотеку классов iostream.
ostream& operator <<(ostream& outs,
                     const Percent& a_percent)
{
    outs << a_percent.value << '%';
    return outs;
}
```

Практические задания

1. Модифицируйте определение класса Money, показанное в листинге 8.7, добавив в него следующие элементы:

   а) операторы <, <=, > и >=, перегруженные для применения к объектам типа Money (См. упражнение 13.);

   б) функцию-член; ниже приводится ее объявление, которое должно входить в определение класса Money. (Определение этой функции должно включать спецификатор Money::.)

   ```
   Money percent(int percent_figure) const;
   // Возвращает значение заданного процента от суммы, хранящейся в вызывающем
   // объекте. Например, если значение аргумента percent_figure равно 10,
   // функция возвращает 10 % суммы, представленной вызывающим объектом.
   ```

   Например, если purse — объект типа Money, представляющий сумму $100,10, то вызов

   ```
   purse.percent(10);
   ```

   вернет 10 % от $100,10, то есть значение типа Money, представляющее сумму $10,01.

2. В упражнении 17 вам предлагалось перегрузить операторы >> и << класса Pairs. Завершите и протестируйте код, созданный для этого упражнения. Напишите используемый по умолчанию конструктор и конструкторы с одним и двумя параметрами типа int. Конструктор с одним параметром должен инициализировать первую переменную пары и присваивать второй переменной значение 0.

   Перегрузите бинарный оператор + для сложения чисел в соответствии со следующим правилом:

   $(a, b) + (c, d) = (a + c, b + d)$

Аналогичным образом перегрузите оператор -.

Перегрузите оператор умножения чисел согласно следующему правилу:

$(a, b) * c = (a * c, b * c)$

Напишите программу для тестирования всех функций-членов и перегруженных операторов класса `Pairs`.

3. В упражнении 18 вам было предложено перегрузить операторы >> и << класса `Percent`. Завершите и протестируйте код, созданный для этого упражнения. Напишите используемый по умолчанию конструктор и конструкторы с одним параметром типа `int`. Перегрузите операторы + и - для сложения и вычитания процентов. Кроме того, перегрузите оператор * для умножения выраженного в процентах значения на целое число.

   Напишите программу для тестирования всех функций-членов и перегруженных операторов класса `Percent`.

4. Определите класс для рациональных чисел. Рациональными называются числа, которые можно представить в виде дроби с целочисленными числителем и знаменателем, например, `1/2`, `3/4`, `64/2` и т. п. (Под `1/2` и подобными представлениями понимаются обычные дроби, а не целочисленное деление в программе на C++.) Представьте рациональные числа как два значения типа `int` — одно для числителя и одно для знаменателя. Назовите класс `Rational`.

   Включите в класс конструктор с двумя аргументами, которые можно использовать для присваивания переменным-членам создаваемого объекта любых допустимых значений. Еще добавьте в класс конструктор с единственным параметром типа `int`, назовите этот параметр `whole_number` и определите конструктор таким образом, чтобы объект инициализировался рациональным числом `whole_number/1`. Кроме того, добавьте конструктор, используемый по умолчанию, инициализирующий объект значением `0` (то есть `0/1`).

   Перегрузите операторы >> и <<. Числа должны вводиться и выводиться в форме `1/2`, `15/32`, `300/401` и т. п. Обратите внимание, что числитель и знаменатель могут содержать знак минус, так что будут возможны входные значения `-1/2`, `15/-32` и `-300/-401` и т. п. Перегрузите операторы ==, <, <=, >, >=, +, -, * и /, чтобы они правильно применялись к типу `Rational`. Напишите программу для тестирования этого класса.

   Подсказка. Два рациональных числа *a/b* и *c/d* равны, когда *a*d* равно *c*b*. Если *b* и *d* — положительные рациональные числа, то *a/b* меньше *c/d* в том случае, когда *a*d* меньше *c*b*. В класс нужно включить функцию для нормализации (сокращения) значений объектов, чтобы после этой операции числитель и знаменатель имели минимальные значения, а знаменатель был положительным. Например, рациональное число `4/-8` в результате нормализации должно быть представлено как `-1/2`.

5. Определите класс для комплексных чисел. Комплексным называется число, представляемое в форме

   $a + b*i$

Здесь (в нашем случае) a и b — числа типа `double`, а i — число, представляющее значение $\sqrt{-1}$. Представьте комплексное число как два значения типа `double`, хранящихся в переменных-членах `real` и `imaginary`. (Переменная-член `imaginary` соответствует числу, умножаемому на i.) Назовите класс `Complex`.

Включите в класс конструктор с двумя параметрами типа `double`, которые можно использовать для присваивания переменным-членам создаваемого объекта любых значений. Кроме того, включите в класс конструктор с единственным параметром типа `double`, назовите этот параметр `real_part` и определите конструктор таким образом, чтобы объект инициализировался значением `real_part + 0*i`. Также добавьте конструктор, используемый по умолчанию, инициализирующий объект значением `0` (то есть `0 + 0*i`).

Перегрузите операторы ==, +, -, *, >> и <<, чтобы они правильно применялись к типу `Complex`. Напишите программу для тестирования этого класса.

Подсказка. Для сложения или вычитания двух комплексных чисел нужно по отдельности выполнить сложение или вычитание над их компонентами (переменными-членами типа `double`). Умножение двух комплексных чисел выполняется по следующей формуле:

$$(a + b*i)*(c + d*i) == (a*c - b*d) + (a*d + b*c)*i$$

Константу i определите так:

```
const Complex i(0, 1);
```

Это и будет описанное выше значение i.

Глава 9 Раздельная компиляция и пространства имен

Те фолианты из моей библиотеки,
Что я превыше герцогства ценю.

Уильям Шекспир

В этой главе будут рассмотрены вопросы, связанные с разделением программы C++ на отдельные части. В разделе 9.1, посвященном раздельной компиляции, рассказывается, как распределить программу между несколькими файлами. Это дает следующие преимущества: при изменении некоторых частей программы вам нужно будет перекомпилировать только ее измененные части, а кроме того, фрагменты программы, выделенные в отдельные файлы, легче повторно использовать в других приложениях.

В разделе 9.2 рассказывается о пространствах имен, с которыми вы познакомились в главе 2. Их применяют с целью обеспечить возможность повторного использования имен классов, функций и других элементов кода путем их уточнения на конкретную реализацию. Пространства имен разделяют весь использующийся в программе код на разделы, в каждом из которых имена должны быть уникальными, но в различных разделах могут повторяться. Фактически это средство локализации имен, более универсальное, чем использование локальных переменных.

9.1. Раздельная компиляция

Ваше «если» — это великий миротворец;
в «если» огромная сила.

Уильям Шекспир

C++ включает средства разделения программы на части, которые можно хранить в отдельных файлах, компилировать раздельно и затем компоновать одной командой или автоматически непосредственно перед запуском программы. Определение

класса (а также связанные с ним определения функций) и применяющую его программу можно поместить в разные файлы. Это позволяет формировать библиотеки классов, предназначенные для использования многими программами. Однажды откомпилированный класс может применяться в разных программах, подобно предопределенным библиотекам (например, библиотекам, подключаемым с помощью заголовочных файлов `iostream` и `cstdlib`). Более того, каждый класс можно разделить на два файла, в одном из которых будет храниться его спецификация, а в другом реализация. Если класс определен согласно рекомендациям, приведенным в нашей книге, и вы изменили только его реализацию, достаточно перекомпилировать лишь файл реализации, остальные же файлы, включая и файлы использующих этот класс программ, не нуждаются ни в изменении, ни в перекомпиляции. В этом разделе мы расскажем о том, как обеспечить такую раздельную компиляцию классов.

Еще раз об абстрактных типах данных

Напомним, что абстрактный тип данных — это класс, определенный таким образом, чтобы его интерфейс и реализация были разделены. Данному правилу должны соответствовàть все создаваемые вами классы. При реализации класса как абстрактного типа нужно отделить спецификацию этого класса от деталей его реализации. Это разделение должно быть настолько полным, чтобы изменение реализации класса не требовало изменения использующих его программ. Для полного разделения интерфейса и реализации класса вы должны следовать трем правилам.

1. Объявите все переменные-члены класса как закрытые.
2. Каждую из основных операций абстрактного типа (класса) определите как открытую функцию-член класса, дружественную функцию, обычную функцию или перегруженный оператор. Объедините определение класса, объявления функций и операторов в одну группу, которая вместе с пояснительными комментариями называется *интерфейсом* абстрактного типа. Комментарии в определении класса и объявлениях функций и операторов должны содержать описания того, как используется каждая функция и оператор.
3. Сделайте реализацию основных операций над абстрактным типом данных недоступной использующему ее программисту. Реализация состоит из определений функций и перегруженных операторов, а также определений вспомогательных функций и других дополнительных элементов.

Чтобы обеспечить максимально надежное соблюдение этих правил в C++, лучше всего поместить интерфейс и реализацию абстрактного типа в отдельные файлы. Как нетрудно предположить, файл, в котором находится интерфейс, называется *файлом интерфейса*, а файл, содержащий реализацию, именуется *файлом реализации*. Детали процесса создания, компиляции и использования этих двух файлов в разных версиях C++ могут немного различаться, но общая схема всегда одинакова. В частности, содержимое файлов во всех системах одинаково, а команды, используемые для их компиляции и компоновки, разные. Процесс разделения интерфейса и реализации абстрактного типа данных на два файла подробно описан в приведенном ниже примере.

Абстрактный тип данных включает закрытые переменные-члены и закрытые функции-члены, использование которых несколько не вписывается в общую концепцию размещения интерфейса и реализации в разных файлах. Дело в том, что открытая часть определения класса входит в состав его интерфейса, тогда как закрытая часть — в состав реализации. А поскольку C++ не позволяет разделять определение класса на два файла, приходится идти на компромисс. Единственно возможным решением здесь является помещение всего определения класса в файл интерфейса, и так как программист, использующий абстрактный тип данных, не имеет доступа к закрытым членам класса, можно по-прежнему считать, что они от него скрыты.

Абстрактный тип данных

Тип данных называется абстрактным, если использующий его программист не имеет доступа к деталям реализации значений и операций этого типа. Все определяемые вами классы должны быть реализованы как абстрактные типы, то есть при их разработке нужно придерживаться ключевого правила: интерфейс и реализацию класса обязательно следует разделить. (Базовые операции, реализованные не как члены класса (в частности, перегруженные операторы) считаются частью абстрактного типа, хотя формально они могут не являться частью определения класса.)

Пример: DigitalTime — отдельно компилируемый класс

В листинге 9.1 приведен файл интерфейса абстрактного типа данных, реализованного в виде класса `DigitalTime`. Значением объекта этого класса является время суток, например 9:30. Интерфейс класса включает только его открытые члены; закрытые члены являются частью реализации, хотя и объявлены в файле интерфейса,— ключевое слово `private:` указывает, что следующие за ним закрытые члены не являются частью открытого интерфейса. Все что нужно знать программисту для использования абстрактного типа `DigitalTime` объясняется в комментарии в начале файла и в комментариях в открытом разделе определения класса. Интерфейс класса говорит программисту, как использовать две версии функции-члена `advance`, конструкторы и перегруженные операторы ==, >> и <<. Функция-член `advance`, перегруженные операторы и оператор присваивания — это все, чем может пользоваться программист для манипулирования объектами и значениями класса. Как указывается в комментарии в начале файла интерфейса, в этом классе используется 24-часовой формат представления времени, поэтому, например, 1:30 после полудня вводится и выводится как 13:30. Эта и другие подробности, необходимые для эффективного использования класса `DigitalTime`, описаны в комментариях, сопровождающих объявления функций-членов.

Интерфейс класса `DigitalTime` мы поместили в файл dtime.h, расширение .h указывает, что это заголовочный файл. Учтите, что файлы интерфейса всегда являются заголовочными, поэтому их имена обязательно имеют названное расширение.

Любая программа, в которой используется класс DigitalTime, должна содержать следующую директиву #include:

```
#include "dtime.h"
```

В директиве #include должно быть указано, является заданный в ней заголовочный файл предопределенным или написан вами. В первом случае его имя заключается в угловые скобки, например <iostream>, во втором случае оно располагается в кавычках, вот так: "dtime.h". Это различие в синтаксисе указывает компилятору, где следует искать требуемый заголовочный файл. Если имя файла заключено в угловые скобки, компилятор ищет его в том каталоге, где хранятся предопределенные заголовочные файлы вашей реализации C++, если же оно задано в кавычках, компилятор осуществляет поиск этого файла в текущем каталоге или в других каталогах, где в вашей системе хранятся заголовочные файлы, определяемые программистом.

Любая программа, в которой используется класс DigitalTime, должна содержать приведенную выше директиву #include с указанием заголовочного файла dtime.h. Этого достаточно, чтобы откомпилировать программу, но недостаточно, чтобы ее запустить. Для выполнения программы нужно написать и откомпилировать определения функций-членов и перегруженных операторов. Данные определения мы поместили в другой файл, называемый файлом реализации. Хотя большинство компиляторов этого не требует, традиционно файлы интерфейса и реализации называют одним и тем же именем. Отличаются они только расширениями, указывающими на их типы. Так, интерфейс нашего абстрактного типа данных мы поместили в файл dtime.h, а его реализацию – в файл dtime.cpp. В разных версиях языка расширение файла реализации может быть различным, поэтому применяйте для файлов реализации то расширение, которым вы обычно пользуетесь для файлов программ C++. Если имена файлов программ оканчиваются на .cxx, используйте это расширение вместо .cpp, если же имена оканчиваются на .cpp, применяйте именно это расширение. Мы используем расширением .cpp, поскольку большинство компиляторов поддерживает его как расширение файлов с исходным кодом C++. Файл реализации класса DigitalTime приведен в листинге 9.2. После выяснения того, как разные файлы нашего абстрактного типа взаимодействуют между собой, мы вернемся к листингу 9.2 и рассмотрим детали определений, имеющихся в этом файле.

Листинг 9.1. Файл интерфейса класса DigitalTime

```
// Заголовочный файл dtime.h. Это ИНТЕРФЕЙС класса DigitalTime.
// Значения данного типа представляют время суток. Они вводятся
// и выводятся в 24-часовом формате, например 9:30 для 9:30
// утра и 14:45 для 2:45 пополудни.
#include <iostream>   // Для определения типов istream и ostream,
                      // использующихся как типы параметров.
using namespace std;

class DigitalTime
{
```

продолжение

Листинг 9.1 (*продолжение*)

```
public:
    friend bool operator ==(const DigitalTime& time1, const DigitalTime& time2);
    // Возвращает true, если значения аргументов time1 и time2 представляют одно
    // и то же время; в противном случае возвращает false.

    DigitalTime(int the_hour, int the_minute);
    // Предусловие: 0 <= the_hour <= 23 и 0 <= the_minute <= 59.
    // Инициализирует объект: the_hour – часы, the_minute – минуты.

    DigitalTime();
    // Инициализирует объект значением времени, равным 0:00 (полночь).

    void advance(int minutes_added);
    // Предусловие: объект содержит значение времени.
    // Постусловие: минуты увеличены на значение аргумента minutes_added.

    void advance(int hours_added, int minutes_added);
    // Предусловие: объект содержит значение времени.
    // Постусловие: значение времени увеличено на
    // hours_added часов и minutes_added минут.

    friend istream& operator >>(istream& ins, DigitalTime& the_object);
    // Перегружает оператор >> для ввода значений типа DigitalTime.
    // Предусловие: если ins – входной файловый поток, он уже
    // соединен с файлом.

    friend ostream& operator <<(ostream& outs, const DigitalTime& the_object);
    // Перегружает оператор << для вывода значений типа DigitalTime.
    // Предусловие: если outs – выходной файловый поток, он уже
    // соединен с файлом.

/* Это часть реализации, а не интерфейса. Ключевое слово private
   указывает, что это не часть открытого интерфейса.*/
private:
    int hour;
    int minute;
};
```

Листинг 9.2. Файл реализации класса DigitalTime

```
// Файл реализации dtime.cpp. (Ваша система может требовать, чтобы
// данный файл имел другое расширение, отличное от .cpp.)
// Это реализация абстрактного типа данных DigitalTime.
// Интерфейс класса DigitalTime находится в заголовочном файле dtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "dtime.h"
using namespace std;

// Эти объявления функций предназначены для использования
// в определении перегруженного оператора ввода >>:

void read_hour(istream& ins, int& the_hour);
// Предусловие: следующими данными в потоке ins являются значения времени
// (формат 9:45 или 14:45).
```

```
// Постусловие: переменной the_hour присвоено количество часов
// из общего значения времени. Двоеточие удалено из входного потока,
// и следующим значением в нем является количество минут.

void read_minute(istream& ins, int& the_minute);
// Считывает из потока ins количество минут, оставшееся после того,
// как с помощью функции read_hour было прочитано количество часов.

int digit_to_int(char c);
// Предусловие: c – одна из цифр от '0' до '9'.
// Возвращает целочисленный символ, представляющий эту цифру; например,
// функция digit_to_int('3') возвращает 3.

bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
{
    return (time1.hour == time2.hour && time1.minute == time2.minute);
}

// Используем библиотеки классов iostream и cstdlib.
DigitalTime::DigitalTime(int the_hour, int the_minute)
{
    if (the_hour < 0 || the_hour > 23 || the_minute < 0 || the_minute > 59)
    {
        cout << "Illegal argument to DigitalTime constructor.";
        exit(1);
    }
    else
    {
        hour = the_hour;
        minute = the_minute;
    }
}

DigitalTime::DigitalTime() : hour(0), minute(0)
{
    // Тело намеренно оставлено пустым.
}

void DigitalTime::advance(int minutes_added)
{
    int gross_minutes = minute + minutes_added;
    minute = gross_minutes%60;

    int hour_adjustment = gross_minutes/60;
    hour = (hour + hour_adjustment)%24;
}

void DigitalTime::advance(int hours_added, int minutes_added)
{
    hour = (hour + hours_added)%24;
    advance(minutes_added);
}

// Используем библиотеку классов iostream.
ostream& operator <<(ostream& outs, const DigitalTime& the_object)
{
    outs << the_object.hour << ':';
    if (the_object.minute < 10)
```

продолжение ⇩

Листинг 9.2 (*продолжение*)

```
        outs << '0';
    outs << the_object.minute;
    return outs;
}

// Используем библиотеку классов iostream.
istream& operator >>(istream& ins, DigitalTime& the_object)
{
    read_hour(ins, the_object.hour);
    read_minute(ins, the_object.minute);
    return ins;
}

int digit_to_int(char c)
{
    return (int(c) - int('0'));
}

// Используем библиотеки классов iostream, cctype и cstdlib.
void read_minute(istream& ins, int& the_minute)
{
    char c1, c2;
    ins >> c1 >> c2;

    if (!(isdigit(c1) && isdigit(c2)))
    {
        cout << "Error illegal input to read_minute\n";
        exit(1);
    }

    the_minute = digit_to_int(c1)*10 + digit_to_int(c2);

    if (the_minute < 0 || the_minute > 59)
    {
        cout << "Error illegal input to read_minute\n";
        exit(1);
    }
}

// Используем библиотеки классов iostream, cctype и cstdlib.
void read_hour(istream& ins, int& the_hour)
{
    char c1, c2;
    ins >> c1 >> c2;
    if ( !( isdigit(c1) && (isdigit(c2) || c2 == ':' ) ) )
    {
        cout << "Error illegal input to read_hour\n";
        exit(1);
    }

    if (isdigit(c1) && c2 == ':')
    {
        the_hour = digit_to_int(c1);
    }
    else   // (isdigit(c1) && isdigit(c2))
    {
```

```
        the_hour = digit_to_int(c1)*10 + digit_to_int(c2);
        ins >> c2;  // discard ':'
        if (c2 != ':')
        {
            cout << "Error illegal input to read_hour\n";
            exit(1);
        }
    }

    if ( the_hour < 0 || the_hour > 23 )
    {
        cout << "Error illegal input to read_hour\n";
        exit(1);
    }
}
```

Листинг 9.3. Файл приложения, в котором используется класс DigitalTime

```
// Файл приложения timedemo.cpp. (Ваша система может потребовать,
// чтобы данный файл имел другое расширение, отличное от .cpp.)
// Эта программа демонстрирует использование класса DigitalTime.
#include <iostream>
#include "dtime.h"
using namespace std;

int main()
{
DigitalTime clock, old_clock;

    cout << "Enter the time in 24 hour notation: ";
    cin >> clock;

    old_clock = clock;
    clock.advance(15);
    if (clock == old_clock)
        cout << "Something is wrong.";
    cout << "You entered " << old_clock << endl;
    cout << "15 minutes later the time will be "
         << clock << endl;

    clock.advance(2, 15);
    cout << "2 hours and 15 minutes after that\n"
         << "the time will be "
         << clock << endl;

    return 0;
}
```

Пример диалога

```
Enter the time in 24-hour notation: 11:15
You entered 11:15
15 minutes later the time will be 11:30
2 hours and 15 minutes after that
the time will be 13:45
```

Для того чтобы в программе можно было использовать класс абстрактного типа DigitalTime, в ней необходима директива

```
#include "dtime.h"
```

Обратите внимание, что эту директиву #include с именем файла интерфейса должны содержать и файл реализации, и файл программы. Второй из них (то есть файл, включающий функцию main), часто называют *файлом приложения*. В листинге 9.3 приведен файл приложения с очень простой программой, предназначенной для демонстрации работы класса DigitalTime.

Точная процедура запуска полной программы, содержащейся в этих трех файлах, зависит от используемой вами системы, но суть ее всегда одинакова — нужно откомпилировать файл реализации и файл приложения, содержащий функцию main. Файл интерфейса, в данном случае dtime.h (см. листинг 9.1), не требует компиляции, так как компилятор считает, что его текст уже содержится в каждом из двух других файлов. Напомним, что файлы реализации и приложения содержат директиву

```
#include "dtime.h"
```

При компиляции программы препроцессор считывает ее и заменяет текстом указанного в ней файла, то есть dtime.h. Поэтому компилятор воспринимает содержимое файла как часть компилируемой программы, и следовательно, данный файл не нуждается в отдельной компиляции. Копирование файла dtime.h является условной процедурой, на самом деле физическое копирование не выполняется, но компилятор действует так, словно содержимое файла dtime.h входит в состав каждого из файлов, содержащих соответствующую директиву #include. Если же заглянуть, скажем, в файл приложения после его компиляции, он останется точно таким же, как до компиляции, — содержимого файла dtime.h там не окажется.

После компиляции файлы реализации и приложения нужно связать, чтобы обеспечит их совместную работу. Такое связывание называется *компоновкой* (linking), а выполняющая его утилита именуется *компоновщиком*. Точная процедура вызова компоновщика зависит от конкретной системы. (Нередко компоновка выполнятся автоматически как часть процесса запуска программы.) После компоновки файлов можно, наконец, запустить программу. Весь этот процесс, возможно, показался вам сложным, но на практике в большинстве систем имеются средства его автоматического или полуавтоматического выполнения, так что, чаще всего, подготовка программы к запуску выполняется быстро и просто.

В листингах 9.1–9.3 приведена одна полная программа, разделенная на части, хранящиеся в разных файлах. Содержимое этих трех файлов можно объединить в один, а затем откомпилировать его и запустить без применения директив #include и компоновки. Возникает закономерный вопрос: затем тратить время на работу с тремя разными файлами? Дело в том, что разделение программы на части имеет несколько преимуществ. Когда определение и реализация класса DigitalTime хранятся отдельно от приложения, этот класс можно использовать во множестве разных программ, не переписывая для каждой из них определение класса. Более того, файл реализации достаточно откомпилировать только один раз независимо от количества программ, в которых будет использоваться класс DigitalTime. Кроме того, есть и другие преимущества. Поскольку интерфейс класса отделен от его

реализации, можно изменить файл реализации, не внося никаких изменений в программы, использующие данный абстрактный тип, причем вам даже не придется перекомпилировать программу. После внесения изменений в файл реализации его нужно перекомпилировать и выполнить повторную компоновку файлов. И хотя экономия времени на перекомпиляции является несомненным достоинством, еще важнее то, что не требуется переписывать код программ — один и тот же класс абстрактного типа данных можно использовать во множестве программ без включения его кода в каждую из них. И для изменения реализации класса не нужно вносить изменения во все эти программы, как было бы, если бы код класса копировался в файл каждой программы, где он используется. Только представьте, каково вносить одни и те же поправки в десятки файлов разных программ, и насколько сложно их согласовывать.

А теперь от организации и использования файлов класса абстрактного типа данных и программы перейдем к более подробному описанию нашего класса из листинга 9.2. Большая часть кода этого класса проста и понятна, но две вещи требуют пояснения. Обратите внимание, что функция-член `advance` перегружена и имеет два определения. Кроме того, в определении перегруженного оператора ввода применяются две вспомогательные функции, `read_hour` и `read_minute`, в которых используется третья вспомогательная функция `digit_to_int`. Давайте подробно рассмотрим эти элементы.

Итак, в состав класса `DigitalTime` (см. листинги 9.1 и 9.2) входят две версии функции-члена `advance`. Одна из них принимает один аргумент — целое число, представляющее количество минут, которое нужно прибавить к текущему времени, представленному объектом. Вторая версия принимает два аргумента — количество часов и количество минут — и тоже увеличивает время объекта согласно этим значениям. Обратите внимание, что в определении функции `advance` с двумя аргументами выполняется вызов функции `advance` с одним аргументом (см. листинг 9.2). Сначала названная функция с двумя аргументами увеличивает время на `hours_added` часов, а затем, вместо того чтобы повторять код функции `advance` с одним аргументом для увеличения времени на `minutes_added` минут, просто вызывает эту функцию. В том что одна версия функции вызывает другую версию этой же функции, нет ничего противоречащего правилам, поскольку две эти версии с точки зрения компилятора являются двумя разными функциями, имеющими одно и то же имя, но разное количество аргументов. Для него эта ситуация ничем не отличается от той, в которой мы могли бы назвать вторую версию функции `another_advance`.

Теперь обратимся к вспомогательным функциям. Функции `read_hour` и `read_minute` посимвольно считывают входные значения, преобразуют их в целые числа и помещают в переменные-члены `hour` и `minute` соответственно. В процессе ввода они считывают число по одной цифре в локальные переменные типа `char`. Этот способ ввода сложнее, чем ввод в переменные типа `int`, но зато позволят детально проверять вводимые данные и в случае их несоответствия ожидаемому формату выводить сообщение об ошибке. В функциях `read_hour` и `read_minute` используется еще одна вспомогательная функция, `digit_to_int`, — такая же, какой мы пользовались в определении класса `Money` (программа в листинге 8.3). Она преобразует цифру в символьном представлении (например `'3'`, в число 3).

Разделение класса на отдельные файлы

Определение класса и реализацию его функций-членов можно поместить в разные файлы. После этого класс можно будет компилировать отдельно от использующей его программы и применять в других программах. Чтобы поместить класс и использующую его программу в три разных файла, необходимо выполнить следующие действия.

1. Поместите определение класса в заголовочный файл, называемый *файлом интерфейса*. Имя заголовочного файла имеет расширение .h. Включите в файл интерфейса объявления функций и перегруженных операторов, реализующих базовые операции класса, но не входящих в его определение. Сопроводите все функции и операторы комментариями, поясняющими, как ими пользоваться.
2. Определения всех функций и перегруженных операторов (членов класса, дружественных и прочих элементов) поместите в другой файл, называемый *файлом реализации*. Включите в него директиву #include с именем файла интерфейса, созданного в предыдущем пункте. Это имя файла должно быть заключено в двойные кавычки, как в следующем примере:

   ```
   #include "dtime.h"
   ```

   Файлам интерфейса и реализации традиционно присваиваются одинаковые имена, но с разными расширениями. Интерфейсный файл имеет расширение .h, а файл реализации — то расширение, которое в данной системе используется для файлов программ C++. Файл реализации компилируется отдельно до его использования в программе.
3. Когда класс потребуется в программе, поместите ее функцию main (и все дополнительные определения функций, объявления констант и переменных и т. п.) в третий файл, называемый *файлом приложения*. Он тоже должен содержать директиву #include с именем файла интерфейса, например:

   ```
   #include "dtime.h"
   ```

   Файл приложения компилируется отдельно от файла реализации. Можно написать любое количество приложений с одной парой файлов интерфейса и реализации класса. Для запуска программы нужно скомпоновать объектный код, сгенерированный в результате компиляции файла приложения, с объектным кодом, сгенерированным в результате компиляции файла реализации класса. (В некоторых системах компоновка может выполняться автоматически или полуавтоматически.)

Многократно используемые компоненты

Классы абстрактных типов данных, разработанные согласно приведенным в этой книге правилам и помещенные в отдельные файлы, представляют собой программные компоненты, которые можно многократно использовать в разных программах. Возможность повторного применения — одна из важнейших задач, стоящих перед программистом при разработке программных компонентов. Компоненты многократного пользования экономят время и силы разработчиков, поскольку их не нужно снова и снова проектировать, кодировать и тестировать для каждого нового приложения. Кроме того, они обычно более надежны, чем однократно используемые компоненты. Ведь, во-первых, на их разработку и отладку можно выделить больше времени, а во-вторых, само применение в разных программах обеспечивает их более полное и разнообразное тестирование. Использование программного компонента много раз и в разных контекстах является одним из лучших способов выявления оставшихся в нем ошибок.

Директива #ifndef

Мы описали метод разделения программы на три файла: интерфейс класса, его реализацию и прикладную часть программы. Однако программа может храниться и более чем в трех файлах. Например, в ней может использоваться несколько классов, каждый из которых хранится в своей паре файлов. Предположим, что у вас имеется программа, разделенная на множество файлов, и часть из них содержит следующую директиву #include:

```
#include "dtime.h"
```

В этом случае одни файлы включают обращения к другим, те, в свою очередь, содержат еще какие-то файлы и т. д. В результате может получиться так, что в файле находится несколько копий dtime.h, логически включенных в него цепочкой директив #include. Однако C++ не позволяет определять один и тот же класс более одного раза, даже если эти определения абсолютно одинаковы. Более того, если один и тот же заголовочный файл используется в разных проектах, почти невозможно проконтролировать, сколько раз в каждый из файлов включается определение класса. Во избежание этой проблемы C++ позволяет пометить раздел кода таким образом, что если этот код уже включен в данный файл, он не включается повторно. Как это сделать, мы покажем на примере.

Следующая директива «определяет» идентификатор DTIME_H:

```
#define DTIME_H
```

Это означает, что препроцессор компилятора помещает идентификатор DTIME_H в некоторый список встреченных компилятором аналогичных идентификаторов. Слово «определяет» не очень точно отражает суть дела, поскольку идентификатор DTIME_H ничего не представляет — он существует сам по себе и просто помещается в список таких же идентификаторов. В C++ имеется другая директива, позволяющая проверить, определен ли идентификатор DTIME_H (или другой заданный в этой директиве идентификатор), и таким образом узнать, обработан ли данный фрагмент кода. Вместо DTIME_H можно использовать любой другой идентификатор, не являющийся ключевым словом C++, но во избежание путаницы выработаны некоторые стандартные соглашения о формировании таких идентификаторов.

Приведенная далее директива проверяет, определен ли идентификатор DTIME_H:

```
#ifndef DTIME_H
```

Если это так, весь код, находящийся между данной директивой и первым вхождением следующей директивы:

```
#endif
```

пропускается компилятором.

(Код, находящийся между директивами #ifndef и #endif, можно прочитать следующим образом: «Если имя DTIME_H не определено, компилятор обрабатывает весь код до следующей директивы #endif». Название директивы #ifndef происходит от англ. if not defined — если не определено. На будущее учтите, что директива #ifdef без буквы n тоже существует.)

А теперь рассмотрим такой код:

```
#ifndef DTIME_H
#define DTIME_H
... Определение класса. ...
#endif
```

Если он присутствует в файле dtime.h, то независимо от количества вхождений в программе директивы

```
#include "dtime.h"
```

класс будет определен только один раз.

При первой обработке директивы

```
#include "dtime.h"
```

будет определен флаг DTIME_H и класс. Когда компилятор вновь встретит эту директиву, он еще раз начнет обработку файла dtime.h, но на этот раз директива

```
#ifndef DTIME_H
```

укажет ему, что нужно пропустить весь код до директивы

```
#endif
```

и класс не будет определен повторно.

В листинге 9.4 приведен тот же заголовочный файл dtime.h, что и в листинге 9.1, однако на этот раз в него включены директивы, препятствующие повторному определению класса. Если содержимое файла dtime.h такое, как в листинге 9.4, то в любом другом файле, прямо или косвенно содержащем более одного вхождения директивы

```
#include "dtime.h"
```

класс DigitalTime будет определен только один раз.

Листинг 9.4. Как избежать многократного определения класса

```
// Заголовочный файл dtime.h. Это ИНТЕРФЕЙС класса DigitalTime.
// Значения данного типа представляют время суток, значения которого вводятся
// и выводится в 24-часовом формате, например 9:30 для 9:30
// утра и 14:45 для 2:45 пополудни.

#ifndef DTIME_H
#define DTIME_H

#include <iostream>
using namespace std;

class DigitalTime
{

... Определение класса DigitalTime такое же, как в листинге 9.1. ...

};

#endif   // DTIME_H
```

Вместо DTIME_H можно применить любой другой идентификатор, но по общепринятому соглашению используется имя файла, записанное прописными буквами с нижним подчеркиванием вместо точки. Если вы будете следовать этому соглашению, другим программистам легче будет понять ваш код, а кроме того, ни вам ни им не придется запоминать имя флага.

Описанные директивы могут применяться и для указания на пропуск любого другого кода в программных и заголовочных файлах, но в данной книге такие примеры рассмотрены не будут.

Совет программисту: определение других библиотек

Для использования раздельной компиляции не обязательно определять класс. Если в программе имеется набор взаимосвязанных функций, которые вы хотели бы превратить в библиотеку, можно поместить их объявления вместе с сопутствующими комментариями в заголовочный файл точно так же, как это делается для интерфейса класса, а определения функций включить в файл реализации. В результате получится библиотека функций, которую подобно классу можно использовать в разных программах.

Упражнения для самопроверки

1. Предположим, вы определили класс абстрактного типа данных и применяете его в программе. Вы намерены поместить класс и программу в разные файлы, как описывалось в этой главе. Укажите, в какой из трех файлов (интерфейса класса, реализации класса и прикладной программы) следует поместить каждый из указанных ниже элементов:

   а) определение класса;

   б) объявление функции, выполняющей операцию абстрактного типа данных, но не являющуюся ни членом класса, ни дружественной к нему функцией;

   в) объявление перегруженного оператора, выполняющего операцию абстрактного типа данных, но не реализованного ни как член класса, ни как дружественная к нему функция;

   г) определение функции, выполняющей операцию абстрактного типа данных, но не являющуюся ни членом класса, ни дружественной к нему функцией;

   д) определение дружественной функции, выполняющей операцию абстрактного типа данных;

   е) определение функции-члена класса;

   ж) определение перегруженного оператора, выполняющего операцию абстрактного типа данных, но не реализованного ни как член класса, ни как дружественная к нему функция;

   з) определение перегруженного оператора, выполняющего операцию абстрактного типа данных и реализованного в виде дружественной функции;

   и) функцию main.

2. Имена каких файлов имеют расширение .h: файла интерфейса класса, файла реализации класса, файла приложения, в котором используется данный класс?
3. Когда класс хранится отдельно от программы, он помещается в двух файлах: интерфейса и реализации. Какой из них нуждается в компиляции — оба, ни один, только один (если да, то какой именно)?
4. Предположим, что класс определен в двух отдельных файлах — интерфейса и реализации — и вы внесли изменения в файл реализации класса. Какие из следующих файлов нуждаются в компиляции: файл интерфейса, файл реализации, файл приложения?
5. Допустим, вы хотите модифицировать реализацию класса `DigitalTime`, приведенную в листингах 9.1 и 9.2, в частности, намерены изменить способ представления времени внутри объектов класса. Вместо того чтобы использовать две закрытые переменные-члены `hour` и `minute`, вы хотите применить одну закрытую переменную `minutes` типа `int`, в которой значения времени будут храниться как количество минут с полуночи (0:00). Например, 1:30 будет храниться как 90 минут, поскольку от полуночи до 1:30 проходит 90 минут. Покажите, как для этого нужно изменить файлы интерфейса и реализации из листингов 9.1 и 9.2. Не приводите эти файлы целиком, только расскажите, какие элементы нуждаются в изменении, и опишите эти изменения в общих чертах.
6. Какая разница между определяемым вами в C++ абстрактным типом данных и определяемым вами классом?

9.2. Пространства имен

Что значит имя? Роза пахнет розой,
Хоть розой назови ее, хоть нет.

Уильям Шекспир

Когда в программе используются классы и функции, написанные разными программистами, вполне возможно, что два разработчика применяют одно и то же имя для различных целей. Чтобы решить эту проблему, в C++ используются пространства имен. *Пространством имен* называется набор определений имен, например определений классов или объявлений переменных.

Пространства имен и директива using

Мы с вами уже применяли пространство имен `std`, которое содержит все имена, определенные в используемых вами стандартных библиотечных файлах (таких, как `iostream` и `cstdlib`). Например, когда вы помещаете в начало файла директиву

```
#include <iostream>
```

в него включаются все определения имен (в частности, потоки `cin` и `cout`) пространства имен `std`. До сих пор единственным известным нам способом указания этого (или любого другого) пространства имен была следующая директива `using`:

```
using namespace std;
```

Чтобы понять, для чего она нужна в программе, следует представить, что произойдет, если ее не включить. Когда программа не содержит директиву using для пространства имен std, в ней можно определить идентификаторы cin и cout со значениями, отличными от их стандартных значений (такое переопределение иногда делают для того, чтобы эти два идентификатора вели себя несколько иначе, чем их обычные версии). Стандартные значения идентификаторов cin и cout определены в пространстве имен std, и без директивы using (или другого ее аналога) ваш код ничего не знает об этом пространстве имен, поэтому единственными определениями потоков cin и cout являются написанные программистом.

Каждый созданный вами фрагмент кода относится к какому-нибудь пространству имен. Если не поместить код в конкретное пространство имен, он будет относиться к *глобальному пространству* имен, для которого не требуется директива using, поскольку оно используется всегда. Можно сказать, что в каждой программе имеется неявная директива using, указывающая, что в ней используется глобальное пространство имен.

Обратите внимание, что в программе одновременно может применяться несколько пространств имен, например, помимо всегда использующегося глобального пространства имен в ней обычно применяется пространство имен std. Ну а что произойдет, если имя определено в двух пространствах имен и вы пользуетесь обоими? В этом случае будет сгенерирована ошибка либо компиляции, либо времени выполнения (в зависимости от ситуации). Одно и то же имя можно определить в двух пространствах имен, но в каждый конкретный момент в программе должно использоваться только одно из них. Хотя это не означает, что вообще нельзя применять два пространства имен в одной программе[1].

Предположим, что в программе имеются два пространства имен, ns1 и ns2, и функция my_function типа void без аргументов, определенная в обоих этих пространствах имен разными способами. Тогда допустим следующий код:

```
{
    using namespace ns1;
    my_function();
}
{
    using namespace ns2;
    my_function();
}
```

В первом вызове используется определение функции my_function из пространства имен ns1, а во втором вызове — определение одноименной функции из пространства имен ns2.

Напомним, что последовательность операторов, объявлений и, возможно, другого кода, заключенная в фигурные скобки, называется блоком. Директива using в начале блока применяется только к этому блоку, поэтому первая директива using применяется к первому блоку, а вторая — ко второму. В таких случаях говорят,

[1] Как будет показано далее в этой главе, существуют способы одновременного использования в программе разных пространств имен, даже если они содержат одно и то же имя, но пока это для нас не важно.

что областью действия пространства имен ns1 является первый блок, а областью действия пространства имен ns2 – второй блок. Именно благодаря возможности ограничения области действия и допустимо использование нескольких пространств имен в одной программе (например, в программе, содержащей два приведенных выше блока).

Директива using часто используется в блоке, составляющем тело определения функции. Если она расположена в начале файла (как мы часто делали до сих пор), то применяется ко всему файлу. Как правило, директиву using помещают либо в начало файла, либо в начало блока.

> **Область действия директивы using**
>
> Областью действия директивы using является блок, в котором она находится (а точнее, его часть от директивы using до конца блока). Если эта директива расположена вне всех блоков, то применяется ко всей следующей за ней части программного кода.

Создание пространств имен

Для того чтобы поместить некоторый код в заданное пространство имен, нужно расположить его после ключевого слова namespace:

```
namespace имя_пространства_имен
{
... Код какой-либо программы. ...
}
```

В результате имена, определенные в коде, будут помещены в пространство имен *имя_пространства_имен*. Для доступа к ним в других частях программы потребуется такая директива using:

```
using namespace имя_пространства_имен;
```

Например, следующий код из программы, приведенной в листинге 9.5:

```
namespace savitch1
{
    void greeting();
}
```

помещает определение функции greeting в пространство имен savitch1.

Листинг 9.5. Использование пространства имен

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

namespace savitch1
{
    void greeting();
}

namespace savitch2
```

```
{
    void greeting();
}

void big_greeting();

int main()
{
/* Для встречающихся здесь имен используются определения из
   пространства имен std и глобального пространства имен. */

    {
      /* Для встречающихся в этом блоке имен используются определения из
         пространств имен savitch2, std и глобального пространства имен. */

        using namespace savitch2;
        greeting();
    }

    {
     /* Для встречающихся в этом блоке имен используются определения из
        пространств имен savitch1, std и глобального пространства имен. */
        using namespace savitch1;
        greeting();
    }

    big_greeting();    return 0;
}

namespace savitch1
{
    void greeting()
    {
        cout << "Hello from namespace savitch1.\n";
    }
}

namespace savitch2
{
    void greeting()
    {
        cout << "Greetings from namespace savitch2.\n";
    }
}

void big_greeting()
{
    cout << "A Big Global Hello!\n";
}
```

Пример диалога

```
Greetings from namespace savitch2.
Hello from namespace savitch1.
A Big Global Hello!
```

Обратившись к листингу 9.5, вы увидите, что определение функции greeting тоже помещено в пространство имен savitch1. Для этого в программе использована еще одна секция namespace:

```
namespace savitch1
{
    void greeting()
    {
        cout << "Hello from namespace savitch1.\n";
    }
}
```

Заметьте, что для одного пространства имен в программе может иметься несколько блоков namespace. В программе листинга 9.5 существуют два блока для пространства имен savitch1 и еще два для пространства имен savitch2.

Каждое имя, определенное в пространстве имен, доступно внутри блока namespace этого пространства имен, но его можно сделать доступным и вне блока. Например, как показано в листинге 9.5, объявление и определение функции в пространстве имен savitch1 можно сделать доступными с помощью следующей директивы using:

```
using namespace savitch1;
```

Упражнения для самопроверки

7. Можно ли в программе из листинга 9.5 вместо имени big_greeting использовать имя greeting?
8. Можно ли в программе из листинга 9.5 добавить такое объявление функции:

   ```
   void greeting(int how_many);
   ```

9. Можно ли для одного пространства имен определить более одного блока namespace?

Уточнение имен

Предположим следующую ситуацию: имеются два пространства имен, ns1 и ns2, и вы хотите воспользоваться функцией fun1, определенной в пространстве имен ns1, и функцией fun2, определенной в пространстве имен ns2, но при этом в обоих пространствах имен определена функция my_function. (Предполагается, что все функции не имеют аргументов, так что о перегрузке речи быть не может.) Использовать директивы

```
using namespace ns1;
using namespace ns2;
```

нельзя из-за конфликтующих определений функции my_function.

Нужно как-то сообщить компилятору, что из пространства имен ns1 нас интересует только функция fun1, а из пространства имен ns2 — только функция fun2. Это можно сделать с помощью следующих строк программы:

```
using ns1::fun1;
using ns2::fun2;
```

Ключевое слово using в форме:

```
using пространство_имен::имя;
```

делает доступным определение элемента *имя* из указанного пространства имен, но остальные его элементы останутся недоступными.

Обратите внимание, что оператор :: вам уже знаком. Так, в программе листинга 9.2 он использовался в определении функции:

```
void DigitalTime::advance(int hours_added, int minutes_added)
{
    hour = (hour + hours_added)%24;
    advance(minutes_added);
}
```

Здесь этот оператор указывает, что определяемая функция advance входит в состав класса DigitalTime, а не является функцией какого-либо другого класса. А строка

```
using ns1::fun1;
```

означает, что мы намерены пользоваться функцией fun1 из пространства имен ns1, а не какого-либо другого пространства имен.

Теперь допустим, что мы хотим один раз (или два-три раза, но не многократно) воспользоваться функцией fun1, определенной в пространстве имен ns1. В этом случае можно сослаться на данную функцию, уточнив ее имя именем пространства имен с помощью оператора ::, вот так:

```
ns1::fun1;
```

Подобным образом можно поступать не только с функциями, но и с другими элементами, определенными в пространстве имен. Такая форма ссылки на пространство имен часто используется при определении типа параметра. Например:

```
int get_number(std::istream input_stream)
...
```

В функции get_number параметр input_stream имеет тип istream, определенный в пространстве имен std. Если из данного пространства имен нам нужно только это имя (или остальные имена пространства имен тоже уточнены ссылкой std::), директива

```
using namespace std;
```

не нужна.

Особенности использования пространств имен (факультативный материал)

Между объявлением с помощью ключевого слова using, допустим, следующего:

```
using std::cout;
```

и директивой using, скажем, такой:

```
using namespace std;
```

имеются важные различия.

Состоят они вот в чем.

1. Объявление с ключевым словом using (такое, как using std::cout;) делает доступным для программного кода только одно имя из заданного в нем пространства имен, тогда как директива using (такая, как using namespace std;) делает доступными все имена этого пространства имен.

2. Объявление с ключевым словом using вводит в программу имя (например, cout), которое нельзя использовать для других целей. Директива using тоже вводит в программу имена из пространства имен, но не фактически.

Первый пункт достаточно очевиден, тогда как второй требует пояснения. Предположим, пространства имен ns1 и ns2 содержат определение функции my_function и не имеют других конфликтов имен. В этом случае такие две строки:

```
using namespace ns1;
using namespace ns2;
```

не вызовут никаких проблем, если имя my_function никогда не будет использоваться в программе в области их действия.

С другой стороны, следующий код:

```
using ns1::my_function;
using ns2::my_function;
```

недопустим, даже если функция my_function никогда не используется в программе, иногда (хотя достаточно редко) этот нюанс может быть важен.

Упражнения для самопроверки

10. Напишите объявление функции wow типа void. У нее два параметра: первый типа speed, определенного в пространстве имен speedway, а второй типа speed, определенного в пространстве имен indy500.

11. Рассмотрим следующие объявления функций из определения класса Money, приведенного в листинге 8.4 главы 8:

    ```
    void input(istream& ins);
    void output(ostream& outs) const;
    ```

    Перепишите их объявления так, чтобы перед ними не требовалась директива

    ```
    using namespace std;
    ```

    (Для этого не нужно заглядывать в программу из листинга 8.4.)

Безымянные пространства имен

В определении класса DigitalTime, приведенном в листингах 9.1 и 9.2, использовались три вспомогательные функции: digit_to_int, read_hour и read_minute. Поскольку они являются частью реализации класса абстрактного типа данных DigitalTime, мы поместили их определения в файл реализации (см. листинг 9.2). Однако после этого они не оказались скрытыми полностью — им следовало бы быть локальными для файла реализации класса DigitalTime, но на самом деле это не так (например, в прикладной программе, использующей класс DigitalTime, нельзя определить

другую функцию с именем digit_to_int или read_hour, или read_minute). Иными словами, оказывается нарушенным принцип сокрытия информации. Для того чтобы действительно скрыть эти вспомогательные функции и сделать локальными для файла реализации класса DigitalTime, их нужно поместить в особое пространство имен, называемое безымянным.

Единица компиляции — это подлежащий компиляции файл (например, файл реализации класса) со всеми файлами, включаемыми в него с помощью директив #include (такими, как заголовочный файл интерфейса класса). Каждая единица компиляции представляет собой *безымянное пространство имен.* Блок namespace для безымянного пространства имен записывается та же, как и для именованного, но без указания имени пространства имен, скажем так:

```
namespace
{
    void sample_function()
    .
    .
    .
}   // Безымянное пространство имен.
```

Все имена, определенные в безымянном пространстве имен, локальны для данной единицы компиляции и могут использоваться вне ее. В качестве примера в листингах 9.6 и 9.7 приведена модифицированная (и окончательная) версия интерфейса и реализации класса DigitalTime. Обратите внимание, что вспомогательные функции (digit_to_int, read_hour и read_minute) определены в безымянном пространстве имен и поэтому локальны для единицы компиляции. Как показано в листинге 9.8, имена из безымянного пространства имен могут повторно использоваться вне его единицы компиляции — здесь имя read_hour применяется для другой функции прикладной программы.

Снова обратившись к файлу реализации, приведенному в листинге 9.7, мы увидим, что вспомогательные функции digit_to_int, read_hour и read_minute используются вне безымянного пространства имен без уточнения именем пространства имен. Любое имя, определенное в безымянном пространстве имен, может без уточнения использоваться в пределах единицы компиляции (поскольку невозможно уточнить что-либо несуществующим именем).

Листинг 9.6. Помещение класса в пространство имен — заголовочный файл

```
// Заголовочный файл dtime.h: это ИНТЕРФЕЙС класса DigitalTime.
// Значения данного типа представляют время суток. Они вводятся
// и выводятся в 24-часовом формате, например 9:30 для 9:30
// утра и 14:45 для 2:45 пополудни.

#ifndef DTIME_H
#define DTIME_H

#include <iostream>
using namespace std;

namespace dtimesavitch
{
```

продолжение

Листинг 9.6 (*продолжение*)

```
/* Первый блок namespace для пространства имен dtimesavitch.
   Второй блок для этого пространства имен находится в файле
   реализации dtime.cpp. */

    class DigitalTime
    {
    ... Определение класса DigitalTime такое же, как в листинге 9.1. ...
    };
}   // Завершение блока пространства имен dtimesavitch.

#endif //DTIME_H
```

Интересно то, как безымянные пространства имен согласуются с правилом C++, запрещающим наличие двух определений имени в одном пространстве имен. В каждой единице компиляции имеется одно безымянное пространство имен. Однако единицы компиляции часто пересекаются. Например, и файл реализации класса, и файл прикладной программы содержат ссылки на файл интерфейса класса (заголовочный), в итоге он входит в состав двух единиц компиляции и двух безымянных пространств имен. Может показаться, что подобные накладки вызывают серьезные проблемы, но на практике этого не происходит. Так, если имя определено в заголовочном файле безымянного пространства имен, его нельзя снова определить в безымянном пространстве имен файла реализации или файла приложения, поэтому никакого конфликта не будет.

Листинг 9.7. Помещение класса в пространство имен — файл реализации

```
// Файл реализации dtime.cpp. (Ваша система может требовать, чтобы
// данный файл имел другое расширение, отличное от .cpp.)
// Это РЕАЛИЗАЦИЯ абстрактного типа данных DigitalTime.
// Интерфейс класса DigitalTime находится в заголовочном файле dtime.h.
#include <iostream>
#include <cctype>
#include <cstdlib>
#include "dtime.h"
using namespace std;

// Первый блок namespace для безымянного пространства имен.
namespace
{
    // Эти объявления функций предназначены для использования
    // в определении перегруженного оператора ввода >>.

    void read_hour(istream& ins, int& the_hour);
    // Предусловие: следующими данными в потоке ins являются значения
    // времени в формате 9:45 или 14:45.
    // Постусловие: переменной the_hour присвоено количество часов из
    // общего значения времени. Двоеточие удалено из входного потока,
    // и следующим значением в нем является количество минут.

    void read_minute(istream& ins, int& the_minute);
    // Считывает из потока ins количество минут, оставшееся после того,
    // как с помощью функции read_hour было прочитано количество часов.

    int digit_to_int(char c);
    // Предусловие: c – одна из цифр от '0' до '9'.
```

```
    // Возвращает целочисленное значение символа, представляющего эту цифру:
    // Например, digit_to_int('3') возвращает 3.
}   // Безымяное пространство имен.

// Блок namespace для пространства имен dtimesavitch. Другой блок namespace
// для этого пространства имен находится в заголовочном файле dtime.h.
namespace dtimesavitch
{
    bool operator ==(const DigitalTime& time1, const DigitalTime& time2)
    ... Остальная часть определения оператора == такая же, как в листинге 9.2. ...

    DigitalTime::DigitalTime()
    ... Остальная часть определения этого конструктора такая же, как в листинге 9.2. ...

    DigitalTime::DigitalTime(int the_hour, int the_minute)
    ... Остальная часть определения этого конструктора такая же, как в листинге 9.2. ...

    void DigitalTime::advance(int minutes_added)
    ... Остальная часть определения функции advance такая же, как в листинге 9.2. ...

    void DigitalTime::advance(int hours_added, int minutes_added)
    ... Остальная часть определения функции advance такая же, как в листинге 9.2. ...

    ostream& operator <<(ostream& outs, const DigitalTime& the_object)
    ... Остальная часть определения оператора << такая же, как в листинге 9.2. ...

    // Используем класс iostream и функции из безымянного пространства имен.
    istream& operator >>(istream& ins, DigitalTime& the_object)
    {
        // Функции, определенные в безымянном пространстве имен локальны для этой единицы
        // компиляции (данного файла и включаемых в него файлов). Они могут использоваться
        // в любом месте файла, но не определены вне его.
        read_hour(ins, the_object.hour);
        read_minute(ins, the_object.minute);
        return ins;
}
} // dtimesavitch

// Еще один блок namespace для безымянного пространства имен.
namespace
{
    int digit_to_int(char c)
    ... Остальная часть определения функции digit_to_int такая же, как в листинге 9.2. ...

    void read_minute(istream& ins, int& the_minute)
    ... Остальная часть определения функции read_minute такая же, как в листинге 9.2. ...

    void read_hour(istream& ins, int& the_hour)
    ... Остальная часть определения функции read_hour такая же, как в листинге 9.2. ...


}   // Безымянное пространство имен.
```

Листинг 9.8. Помещение класса в пространство имен — файл прикладной программы

```
// Файл приложения timedemo.cpp.
// В этой программе демонстрируется использование класса DigitalTime.
#include <iostream>
#include "dtime.h"
```

продолжение ⇨

Листинг 9.8 (*продолжение*)

```
... Если поместить сюда директивы using, поведение программы не изменится. ...
void read_hour(int& the_hour);

int main()
{
    using namespace std;

    using namespace dtimesavitch;

    int the_hour;
    read_hour(the_hour); // Это не та функция read_hour, которая определена
                         // в файле реализации dtime.cpp (см. листинг 9.7).

    DigitalTime clock(the_hour, 0), old_clock;

    old_clock = clock;
    clock.advance(15);
    if (clock == old_clock)
        cout << "Something is wrong.";
    cout << "You entered " << old_clock << endl;
    cout << "15 minutes later the time will be "
         << clock << endl;

    clock.advance(2, 15);
    cout << "2 hours and 15 minutes after that\n"
         << "the time will be "
         << clock << endl;

    return 0;
}

void read_hour(int& the_hour)
{
    using namespace std;

    cout << "Let's play a time game.\n"
         << "Let's pretend the hour has just changed.\n"
         << "You may write midnight as either 0 or 24,\n"
         << "but, I will always write it as 0.\n"
         << "Enter the hour as a number (0 to 24): ";
    cin >> the_hour;
    if (the_hour == 24)
        the_hour = 0;
}
```

Пример диалога

```
Let's play a time game.
Let's pretend the hour has just changed.
You may write midnight as either 0 or 24,
but I will always write it as 0.
Enter the hour as a number (0 to 24): 11
You entered 11:00
15 minutes later the time will be 11:15
2 hours and 15 minutes after that
the time will be 13:30
```

Совет программисту: выбор имени для пространства имен

В имена создаваемых вами пространств имен желательно включать какую-нибудь уникальную строку (например, вашу фамилию), чтобы свести к минимуму вероятность их совпадения с именами пространств имен, разработанных другими программистами. Если несколько человек пишут код для одного проекта, важно чтобы разные пространства имен были названы по-разному, иначе может оказаться, что в одной и той же области видимости у них будет несколько определений одних и тех же имен. Именно поэтому мы включили в имя пространства имен `dtimesavitch` в листинге 9.7 слово `savitch`.

Безымянные пространства имен

С помощью безымянного пространства имен можно сделать определение идентификатора локальным для единицы компиляции. Каждая единица компиляции имеет одно безымянное пространство имен. Все определенные в нем идентификаторы локальны для этой единицы компиляции. Чтобы включить определение в безымянное пространство имен, нужно добавить его в блок `namespace` без имени пространства имен:

```
namespace
{
    определение_1
    определение_2
          .
          .
          .
    определение_n
}
```

Любое имя в безымянном пространстве имен можно использовать без уточнителя в любом месте единицы компиляции. Полный пример приведен в листингах 9.6–9.8.

Ловушка: не путайте глобальное и безымянное пространства имен

Не путайте глобальное и безымянное пространства имен. Если определение имени не включено в блок `namespace`, оно относится к глобальному пространству имен. Для того чтобы включить определение имени в безымянное пространство имен, нужно поместить его в блок `namespace`, начинающийся следующим образом:

```
namespace
{
```

Доступ к именам глобального и безымянного пространств имен возможен без уточнителей. Однако имена первого из них имеют глобальную область видимости (все файлы программы), тогда как имена второго видимы только в единице компиляции.

При написании кода это мнимое сходство двух пространств имен не вызывает особых проблем, поскольку обычно имена глобального пространства представляют как не относящиеся ни к одному из пространств имен (хотя технически это не верно). А вот при чтении кода их легко спутать.

Упражнения для самопроверки

12. Будет ли программа, приведенная в листинге 9.8, вести себя иначе, если заменить следующую директиву using:

```
using namespace dtimesavitch;
```

такой строкой

```
using dtimesavitch::DigitalTime;
```

13. Что выведет следующая программа:

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

namespace sally
{
    void message();
}

namespace
{
    void message();
}

int main()
{
    {
        message();
        using sally::message;
        message();
    }
    message();
    return 0;
}

namespace sally
{
    void message()
    {
        cout << "Hello from Sally.\n";
    }
}

namespace
{
    void message()
    {
        cout << "Hello from unnamed.\n";
    }
}
```

14. В листинге 9.7 приведены два блока namespace: один для объявлений вспомогательных функций и один для их определений. Можно ли удалить блок namespace для объявлений вспомогательных функций (оставив сами объявления)? Если да, то как это сделать?

Резюме

- В C++ абстрактные типы данных представляют как классы с закрытыми переменными-членами. Операции этих типов реализуют в виде открытых функций-членов класса, а также функций, не являющихся членами класса и перегруженными операторами.
- Абстрактный тип данных можно определить как класс и поместить его определение и реализацию функций-членов в разные файлы. Такой класс можно компилировать отдельно от программ, в которых он используется, и применять во множестве разных программ.
- Пространство имен — это набор определений имен, таких как определения классов и объявления переменных.
- Существует три способа использования идентификатора из пространства имен: с помощью директивы `using` сделать все имена этого пространства имен доступными; с помощью объявления с ключевым словом `using` сделать доступным один идентификатор из этого пространства имен; уточнить идентификатор именем пространства имен с помощью оператора `::`.
- Для того чтобы включить определение идентификатора в пространство имен, нужно включить его в блок `namespace` этого пространства имен.
- Безымянное пространство имен можно использовать, чтобы определить имя, локальное для единицы компиляции.

Ответы к упражнениям для самопроверки

1. Элементы, приведенные в пунктах а), б), в), следует поместить в файл интерфейса, элементы, перечисленные в пунктах г)–з), — в файл реализации (все определения операций абстрактного типа включаются в этот файл), а последний элемент, то есть функцию `main`, — в файл прикладной программы.
2. Расширением .h завершается имя файла интерфейса.
3. В компиляции нуждается только файл реализации. Файл интерфейса не компилируется.
4. В перекомпиляции нуждается только файл реализации, однако требуется повторная компоновка файлов.
5. Из файла интерфейса, приведенного в листинге 9.1, нужно удалить закрытые переменные-члены `hour` и `minute` и заменить их переменной-членом `minutes` (с буквой *s* в конце), больше никаких изменений в этот файл вносить не нужно. В файле реализации следует изменить определения всех конструкторов и других функций-членов, а также определения перегруженных операторов, чтобы они поддерживали новый способ хранения значений времени. (В данном случае не требуется модификация ни одной из вспомогательных функций

read_hour, read_minute и digit_to_int, но в каком-нибудь другом классе изменение внутреннего представления данных может потребовать модификации и вспомогательных функций или даже полного пересмотра реализации класса.) Например, определение перегруженного оператора >> небходимо изменить следующим образом:

```
istream& operator >>(istream& ins, DigitalTime& the_object)
{
    int input_hour, input_minute;
    read_hour(ins, input_hour);
    read_minute(ins, input_minute);
    the_object.minutes = input_minute + 60*input_hour;
    return ins;
}
```

Прикладные файлы программы, использующей данный класс, в модификации не нуждаются. Однако, поскольку изменен файл интерфейса (и файл реализации), придется перекомпилировать все файлы приложения и, конечно, файл реализации класса.

6. Краткий ответ таков: абстрактный тип данных — это просто класс, в котором интерфейс отделен от реализации. Кроме того, описывая класс как абстрактный тип, мы считаем некоторые функции, реализующие базовые операции над объектами этого класса (например, перегруженные операторы), частью абстрактного типа, даже если формально они таковой не являются.

7. Нет. Если заменить имя big_greeting именем greeting, в глобальном пространстве имен окажется определение имени greeting. Но существуют части программы, для которых одновременно доступны все определения имен в пространстве имен savitch1 и все определения имен в глобальном пространстве имен. В этих частях программы окажется два разных определения функции

```
void greeting();
```

8. Да, это дополнительное определение не вызовет никаких проблем, поскольку перегрузка допускается всегда. Например, когда доступны пространство имен savitch1 и глобальное пространство имен, функция greeting будет перегружена. В предыдущем упражнении проблема состояла в том, что некоторым фрагментам кода оказывалось доступно по два определения функции greeting с одним и тем же списком параметров.

9. Да, пространство имен может содержать любое количество блоков namespace. Так, следующие блоки пространства имен savitch1:

```
namespace savitch1
{
    void greeting();
}

namespace savitch1
{
    void greeting()
    {
```

```
        cout << "Hello from namespace savitch1.\n";
    }
}
```

определены в программе листинга 9.5.

10. `void wow(speedway::speed s1, indy500::speed s2);`

11. ```
    void input(std::istream& ins);
    void output(std::ostream& outs) const;
    ```

12. Программа будет вести себя точно так же.

13. ```
    Hello from unnamed.
    Hello from Sally.
    Hello from unnamed.
    ```

14. Да, блок для объявлений вспомогательных функций можно удалить, если эти функции объявляются до их использования. Скажем, можно удалить пространство имен с объявлениями вспомогательных функций и переместить блок с их определениями, расположив его непосредственно перед блоком для пространства имен `dtimesavitch`.

Практические задания

1. Добавьте в класс `DigitalTime`, определенный в листингах 9.1 и 9.2, следующую функцию-член:

   ```
   void DigitalTime::interval_since(const DigitalTime& a_previous_time,
             int& hours_in_interval, int& minutes_in_interval) const
   ```

   Она вычисляет интервал времени между двумя значениями типа `DigitalTime`, одним из которых является вызывающий объект (в нем задается время конца интервала), а второе задано в первом аргументе функции (в нем указывается время начала интервала). В качестве примера рассмотрим следующий код:

   ```
   DigitalTime current(5, 45), previous(2, 30);
   int hours, minutes;
   current.interval_since(previous, hours, minutes);
   cout << "The time interval between " << previous
        << " and " << current << endl
        << "is " << hours << " hours and "
        << minutes << " minutes.\n";
   ```

   В программе, использующей новую версию класса `DigitalTime`, этот код выведет такой текст:

   ```
   The time interval between 2:30 and 5:45
   is 3 hours and 15 minutes.
   ```

   Время, заданное в первом аргументе функции, может быть больше времени, заданного в вызывающем объекте. В этом случае считается, что время, заданное в первом аргументе функции, относится к предыдущему дню. Напишите программу для тестирования новой версии класса `DigitalTime`.

2. Полностью выполните упражнение 5, напишите весь код класса абстрактного типа данных, включая файлы интерфейса и реализации. Напишите также программу для тестирования класса.
3. Переделайте (или выполните с нуля) задание 4 из главы 8. Определите класс абстрактного типа данных в отдельных файлах, чтобы его можно было компилировать отдельно.
4. Переделайте (или выполните с нуля) задание 5 из главы 8. Определите класс абстрактного типа данных в отдельных файлах, чтобы его можно было компилировать отдельно.

Глава 10 Массивы

Рассуждать, не имея данных, — огромная ошибка.
Артур Конан Дойл

Массивы — это особый тип данных, предназначенный для обработки однотипной информации, например списка значений температуры или перечня имен. В этой главе рассказывается, как определяются и применяются массивы в языке C++, и описываются основные методы разработки алгоритмов и программ с их использованием.

10.1. Основные понятия

Предположим, вы хотите написать программу, которая запрашивает у пользователя пять оценок за выполненные студентами тесты, после чего производит с введенными значениями некоторые операции. Например, находит самую высокую оценку, которая будет обнаружена только после ввода всех пяти оценок студента, и выводит разницу между ней и каждой из остальных оценок. Чтобы можно было отыскать самую высокую, все пять оценок должны храниться в памяти. Для этого требуется структура данных, эквивалентная пяти переменным типа `int`. Можно, конечно, использовать пять отдельных переменных типа `int`, но с ними неудобно работать, и такое решение имеет очень ограниченную область применения, так как годится для нескольких значений, но совсем не подходит для работы с сотней оценок. Самым подходящим решением в данном случае является массив. *Массив* — это аналог некоторого количества переменных одного типа с унифицированными именами, объявляемый с помощью одной простой строки программного кода. Так, если хранить пять оценок в массиве, названном `score`, идентификаторами пяти переменных, каждая из которых хранит свою конкретную оценку, будут `score[0]`, `score[1]`, `score[2]`, `score[3]` и `score[4]`. Все он имеют одно и то же имя — `score` — и разные индексы, помещенные в квадратные скобки.

Объявление массивов и доступ к их элементам

В C++ массив, состоящий из пяти переменных типа `int`, объявляется так:

```
int score[5];
```

Это объявление подобно объявлению пяти переменных типа int с именами:

score[0], score[1], score[2], score[3], score[4]

Переменные, составляющие массив, называются *индексированными переменными* или *элементами массива*. Доступ к ним осуществляется несколькими способами. Номер элемента массива, заданный в квадратных скобках, называется его *индексом*. В C++ элементы массива индексируются, начиная с 0, а не с 1 или какого-то другого числа. Число элементов массива называется его *объявленным размером* или просто *размером массива*. При объявлении массива его размер задается в квадратных скобках после имени. Таким образом, элементы массива нумеруются, начиная с 0 и заканчивая целым числом, на единицу меньшим размера массива.

В нашем примере элементы массива имеют тип int, но в общем случае они могут быть любого типа. Для того чтобы объявить массив с элементами типа double, нужно просто использовать в его объявлении ключевое слово double вместо int. Но учтите, что в одном массиве не могут храниться элементы разных типов. Тип данных, который имеют элементы массива, называется его *базовым типом*. Таким образом, в нашем примере объявляется массив score с базовым типом int.

Массивы можно объявлять в одном операторе объявления вместе с переменными обычных типов. Скажем, следующий оператор:

```
int next, score[5], max;
```

объявляет две переменные типа int (next и max) и один массив с базовым типом int (score).

Элемент массива (например, score[3]) может использоваться везде, где допускается применение простых переменных типа int.

Не путайте значение в квадратных скобках, которое следуют сразу за именем массива, со значениями в квадратных скобках в других местах программы. Когда квадратные скобки используются в объявлении массива, допустим, в таком:

```
int score[5];
```

заключенное в них число определяет количество содержащихся в массиве элементов. В любом другом месте программы в квадратных скобках после имени массива задается индекс необходимого элемента, и вся конструкция представляет собой обращение к этому элементу. Например, score[0] — первый элемент массива score, а score[4] — его последний элемент.

Номер в квадратных скобках не обязательно задавать с помощью константы. Это может быть любое выражение, возвращающее целочисленное значение от 0 до числа, на единицу меньшего, чем размер массива. В приведенном фрагменте кода:

```
int n = 2;
score[n+1] = 99;
```

элементу массива score[3] присваивается значение 99.

Хотя конструкции score[3] и score[n+1] выглядят по-разному, обе они указывают на один и тот же элемент массива, так как в данном случае выражение n+1 возвращает значение 3. Элемент массива (score[i]) идентифицируется значением индекса (значением i), поэтому можно написать программу, выполняющую следующее:

«произвести такие-то действия с i-ым элементом массива», где значение i вычисляется программой. Программа, приведенная в листинге 10.1, считывает оценки студентов и обрабатывает их так, как рассказывалось в начале главы.

Листинг 10.1. Программа, в которой используется массив

```
// Считывает 5 оценок и показывает, насколько
// каждая из них отличается от наивысшей оценки.
#include <iostream>

int main()
{
    using namespace std;
    int i, score[5], max;

    cout << "Enter 5 scores:\n";
    cin >> score[0];
    max = score[0];
    for (i = 1; i < 5; i++)
    {
        cin >> score[i];
        if (score[i] > max)
            max = score[i];
      // max – это максимальное из значений score[0], ..., score[i].
    }

    cout << "The highest score is " << max << endl
         << "The scores and their\n"
         << "differences from the highest are:\n";
    for (i = 0; i < 5; i++)
        cout << score[i] << " off by "
             << (max-score[i]) << endl;

    return 0;
}
```

Пример диалога

```
Enter 5 scores:
6 9 2 10 6
The highest score is 10
The scores and their
differences from the highest are:
6 off by 4
9 off by 1
2 off by 8
10 off by 0
6 off by 4
```

Совет программисту: используйте для работы с массивами цикл for

В листинге 10.1 на примере использования второго цикла for:

```
for (i = 0; i < 5; i++)
        cout << score[i] << " off by "
             << (max-score[i]) << endl;
```

показан типичный способ перебора элементов массива. Этот цикл идеально подходит для обработки массивов.

Ловушка: элементы массивов всегда нумеруется начиная с нуля

Индексы массивов всегда начинаются с числа 0 и оканчиваются целым числом, на единицу меньшим размера массива.

Совет программисту: задавайте размер массивов с помощью определенных в программе констант

Вернемся к программе, приведенной в листинге 10.1, — она используется только для группы, в которой учится в точности пять студентов. Однако на самом деле количество студентов в группе может быть различным. Чтобы программа приобрела гибкость, размеры массивов в ней можно задавать с помощью именованных констант. Например, перепишем программу листинга 10.1, используя константу

```
const int NUMBER_OF_STUDENTS = 5;
```

Тогда строка с объявлением массива будет такой:

```
int i, score[NUMBER_OF_STUDENTS], max;
```

Понятно, что везде, где в программе встречается размер массива, значение 5, его необходимо заменить именем константы — NUMBER_OF_STUDENTS. После внесения этих изменений адаптировать программу к другому количеству студентов будет очень легко: достаточно изменить значение константы NUMBER_OF_STUDENTS в единственном месте программы.

Учтите, что в объявлении массива его размер нельзя задавать с помощью переменной:

```
cout << "Enter number of students:\n";
cin >> number;
int score[number];
// НЕ ДОПУСКАЕТСЯ БОЛЬШИНСТВОМ КОМПИЛЯТОРОВ
```

Это позволяют делать лишь некоторые компиляторы, но даже если вы применяете такой компилятор, все равно избегайте использования переменной для задания размера массива, чтобы написанные вами программы были максимально переносимыми. (В главе 12 рассказывается о другом типе массивов, размер которых может определяться во время выполнения программы, а не во время ее компиляции.)

Расположение массивов в памяти

Прежде чем говорить о том, в каком виде массивы хранятся в памяти компьютера, разберемся, как в ней представлена обычная переменная типа int или double. Память компьютера состоит из последовательности нумерованных ячеек, называемых байтами. Номер байта — это его адрес. Простая переменная представляет собой

небольшую область памяти, состоящую из некоторого количества последовательных байтов, которое зависит от типа переменной. Таким образом, простая переменная определяется в памяти двумя характеристиками: адресом первого байта переменной в памяти (говоря об адресе переменной, мы имеем в виду именно этот адрес) и типом переменной, задающим количество необходимых для ее хранения байтов. Когда программа сохраняет значение в переменной, она помещает его (закодированное в виде последовательности нулей и единиц) в ту область памяти, которая выделена для данной переменной. А когда переменная передается функции по ссылке в качестве аргумента, функция получает адрес этой переменной.

Ну а теперь вернемся к массивам. Компьютер хранит элементы массива точно так же, как и обычные переменные. Но есть одно отличие: элементы массива располагаются в памяти точно друг за другом и занимают в ней непрерывную область. В качестве примера рассмотрим следующий массив:

```
int a[6];
```

При его объявлении компьютер резервирует память, достаточную для хранения шести переменных типа `int`, всегда помещая элементы массива в памяти один за другим. Затем он запоминает адрес только первого элемента массива — `a[0]`. Когда программе требуется адрес какого-нибудь другого элемента, компьютер вычисляет его на основании адреса элемента `a[0]`. Так, для получения адреса элемента `a[3]` компьютер берет адрес элемента `a[0]`, представляющий собой число, и прибавляет к нему количество байтов, занимаемое тремя переменными типа `int`. Схема адресации массивов показана на рис. 10.1.

Многие особенности массивов C++ основываются именно на способе их расположения в памяти.

Объявление массива

Синтаксис

имя_типа имя_массива`[`*объявленный_размер*`];`

Данный оператор объявляет массив из *объявленный_размер* элементов, к которым можно обращаться при помощи идентификаторов от *имя_массива*`[0]` до *имя_массива*`[`*объявленный_размер*`-1]`. Все элементы массива имеют тип данных *имя_типа*.

Пример

```
int big_array[100];
double a[3];
double b[5];
char grade[10], one_grade;
```

Здесь массив `a` состоит из элементов `a[0]`, `a[1]` и `a[2]` типа `double`, а массив `b` включает элементы `b[0]`, `b[1]`, `b[2]`, `b[3]` и `b[4]` того же типа.

Массивы можно объявлять в одном операторе с объявлениями простых переменных того же типа, что и базовый тип массива. Так, в приведенном примере массив `grade` объявляется в одной строке с переменной `one_grade`.

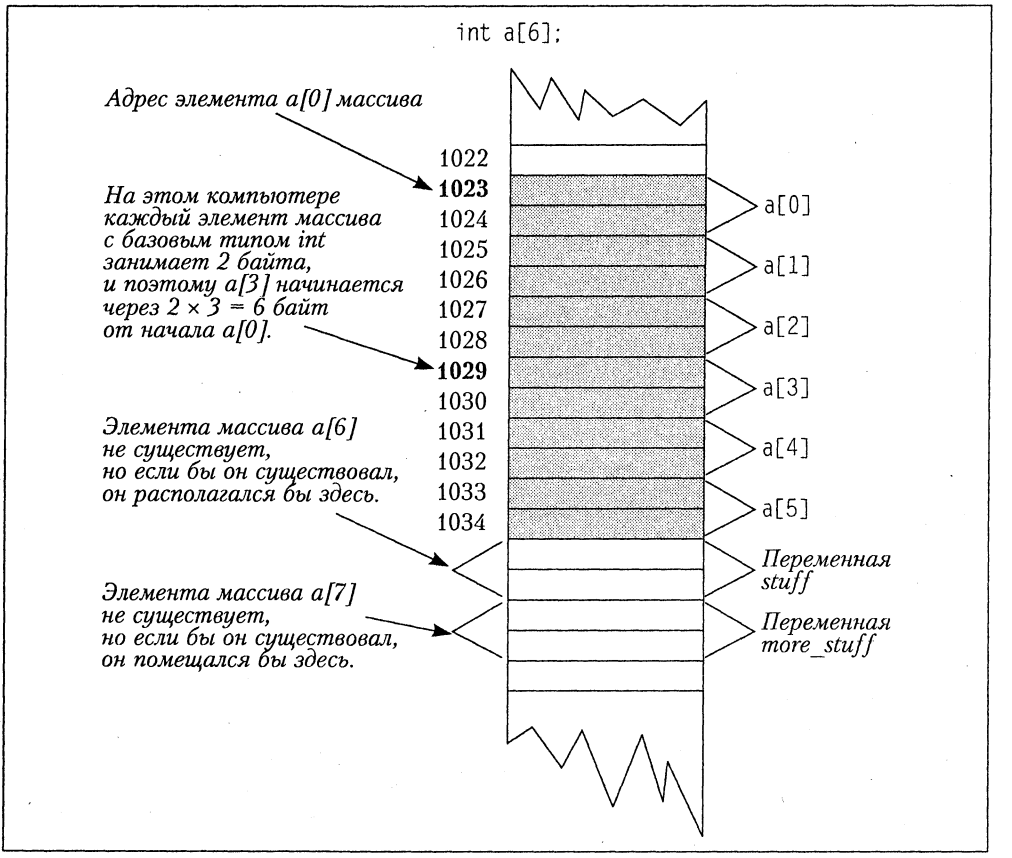


Рис. 10.1. Элементы массива в памяти компьютера

Ловушка: выход индекса массива за допустимые пределы

Одной из наиболее распространенных ошибок в программировании является попытка обращения к элементу массива с несуществующим индексом. Рассмотрим следующее объявление массива:

```
int a[6];
```

Когда программа обращается к элементам массива `a`, то выражение, используемое для определения индекса, должно возвращать целое число от `0` до `5`. Например, если программа выполняет некоторую операцию с элементом массива `a[i]`, переменная `i` должна содержать одно из шести значений: `0`, `1`, `2`, `3`, `4` или `5`. Когда в `i` содержится любое другое значение, выполнение этой операции может привести к ошибке. Если результатом вычисления индексного выражения оказывается индекс не из числа определяемых объявлением массива, такой индекс называют *выходящим за пределы допустимого диапазона* или *недопустимым*. В большинстве систем результатом использования недопустимого индекса массива являются неправильные,

зачастую разрушительные действия программы, причем без вывода какого-либо предупреждения пользователю.

В качестве примера предположим, что в программе содержится приведенное выше объявление массива, а также следующий код:

```
a[i] = 238;
```

Далее допустим, что переменная `i` содержит значение 7. Программа действует так, словно `a[7]` — реально существующий элемент массива. При этом вычисляется физический адрес, по которому он должен располагаться, и по вычисленному адресу в память помещается значение 238. Однако на самом деле элемента массива `a[7]` не существует, и память, в которую записано число 238, может быть закреплена за какой-нибудь переменной (или элементами другого массива), например, `more_stuff`. Таким образом, данное значение окажется измененным, чего программист совершенно не предполагает. Эту ситуацию иллюстрирует рис. 10.1.

Выход за границы массива чаще всего происходит на первой или последней итерации цикла, в котором этот массив обрабатывается. Поэтому будьте особенно внимательны при написании циклов для работы с массивами.

Инициализация массивов

Массив можно инициализировать при его объявлении. В операторе инициализации начальные значения элементов массива заключаются в фигурные скобки и перечисляются по порядку через запятую, как здесь:

```
int children[3] = {2, 12, 1};
```

Это объявление эквивалентно такому коду:

```
int children[3];
children[0] = 2;
children[1] = 12;
children[2] = 1;
```

Если количество значений в списке в фигурных скобках меньше количества элементов массива, инициализация выполняется, начиная с первого элемента. Последние элементы, которым не хватило начальных значений, инициализируются нулевыми значениями базового типа массива. В частности, элементы целочисленного массива инициализируются нулями. Однако в тех случаях, когда массив объявляется без явной инициализации, он вообще не будет инициализирован. То же самое касается и любых переменных, объявленных в определениях функций, включая функцию `main`. Хотя элементы массивов и переменные иногда инициализируются значением 0 автоматически, рассчитывать на это не следует.

Если массив явно инициализируется в объявлении, его размер можно не указывать — в этом случае он будет установлен автоматически в соответствии с количеством заданных значений. Например, такое объявление:

```
int b[] = {5, 12, 11};
```

эквивалентно следующему:

```
int b[3] = {5, 12, 11};
```

Упражнения для самопроверки

1. Чем отличаются выражения `int a[5];` и `a[4]` и что означают их составляющие `[5]` и `[4]`?
2. Для объявления массива

   ```
   double score[5];
   ```

   укажите следующее:

   а) имя массива;

   б) базовый тип данных;

   в) объявленный размер массива;

   г) диапазон значений индекса;

   д) один из элементов массива.
3. Допущены ли ошибки в следующих объявлениях:

   а) `int x[4] = { 8, 7, 6, 4, 3 };`

   б) `int x[] = { 8, 7, 6, 4 };`

   в)
   ```
   const int SIZE = 4;
   int x[SIZE];
   ```

   и если да, то какие?
4. Что выведет код

   ```
   char symbol[3] = {'a', 'b', 'c'};
   for (int index = 0; index < 3; index++)
   cout << symbol[index];
   ```
5. Что выведет код

   ```
   double a[3] = {1.1, 2.2, 3.3};
   cout << a[0] << " " << a[1] << " "
        << a[2] << endl;
   a[1] = a[2];
   cout << a[0] << " " << a[1] << " "
        << a[2] << endl;
   ```
6. Что выведет код

   ```
   int i, temp[10];
   for (i = 0; i < 10; i++)
       temp[i] = 2*i;
   for (i = 0; i < 10; i++)
       cout << temp[i] << " ";
   cout << endl;
   for (i = 0; i < 10; i = i+2)
       cout << temp[i] << " ";
   ```
7. Какие ошибки допущены в следующем фрагменте кода:

   ```
   int sample_array[10];
   for (int index = 1; index <= 10; index++)
       sample_array[index] = 3*index;
   ```
8. Предположим, что элементы массива в программе должны быть упорядочены таким образом:

   `a[0]` ≤ `a[1]` ≤ `a[2]` ≤ ...

Однако мы хотим, чтобы программа на всякий случай проверила, так ли это, и если какой-нибудь элемент окажется не на своем месте, вывела соответствующее сообщение. За вывод данного сообщения отвечает код:

```
double a[10];

... Здесь располагается код, заполняющий массив. ...

for (int index = 0; index < 10; index++)
    if (a[index] > a[index+1])
        cout << "Array elements " << index << " and "
             << (index+1) << " are out of order.";
```

но он содержит ошибку. Что это за ошибка?

9. Напишите программный код на языке C++, заполняющий массив двадцатью значениями типа int, введенными с клавиатуры. Всю программу писать не нужно, но обязательно приведите объявления массива и всех необходимых переменных.
10. Предположим, что в программе имеется следующее объявление массива:

    ```
    int your_array[7];
    ```

    Кроме того, допустим, что в вашей реализации языка C++ переменным типа int выделяется по два байта памяти. Сколько памяти займет этот массив во время выполнения программы? Если предположить, что для элемента массива your_array[0] система выделит память по адресу 0, каким будет адрес элемента массива your_array[3]?

10.2. Массивы и функции

В качестве аргументов функций можно использовать как отдельные элементы массивов, так и целые массивы. Сначала мы рассмотрим как передаются отдельные элементы.

Элементы массива в качестве аргументов функций

Элемент массива может быть аргументом функции точно так же, как и любая переменная.

Предположим, что программа содержит объявление

```
int i, n, a[10];
```

Если функция my_function принимает один аргумент типа int, то ее вызов

```
my_function(n);
```

является допустимым.

Поскольку элемент массива полностью аналогичен переменной типа int, например такой, как переменная n, допустимым является приведенный ниже вызов:

```
my_function(a[3]);
```

Теперь рассмотрим следующий вызов:

```
my_function(a[i]);
```

Если переменная i содержит значение 3, аргументом функции будет значение a[3]. Если же переменная i содержит, например 0, то приведенный вызов эквивалентен такому:

```
my_function(a[0]);
```

Выражение индекса вычисляется до вызова функции, благодаря чему всегда известно, какой из элементов массива должен быть передан в эту функцию.

В листинге 10.2 приведен простой пример использования элементов массива в качестве аргументов функции. Программа выделяет пять дополнительных дней отпуска каждому из трех сотрудников маленькой фирмы. Обратите внимание на функцию adjust_days, у которой имеется параметр old_days типа int. В теле функции main она вызывается с аргументом vacation[number] для разных значений переменной number. Заметьте, что в определении формального параметра old_days нет ничего особенного. Это просто параметр типа int — базового типа массива vacation. В листинге программы 10.2 элементы массива передаются по значению. Однако все сказанное равно относится и к передаче элементов массива по ссылке.

Листинг 10.2. Элемент массива в качестве аргумента

```
// Иллюстрирует использование элемента массива
// в качестве аргумента функции. Увеличивает количество
// дней отпуска каждого сотрудника на 5.
#include <iostream>

const int NUMBER_OF_EMPLOYEES = 3;

int adjust_days(int old_days);
// Возвращает old_days плюс 5.

int main()
{
    using namespace std;
    int vacation[NUMBER_OF_EMPLOYEES], number;

    cout << "Enter allowed vacation days for employees 1"
         << " through " << NUMBER_OF_EMPLOYEES
         << ":\n";
    for (number = 1;
                number <= NUMBER_OF_EMPLOYEES; number++)
        cin >> vacation[number-1];

    for (number = 0; number < NUMBER_OF_EMPLOYEES; number++)
        vacation[number] = adjust_days(vacation[number]);

    cout << "The revised number of vacation days are:\n";
    for (number = 1;
                number <= NUMBER_OF_EMPLOYEES; number++)
        cout << "Employee number " << number
             << " vacation days = " << vacation[number-1]
             << endl;
```

```
    return 0;
}

int adjust_days(int old_days)
{
    return (old_days+5);
}
```

Пример диалога

```
Enter allowed vacation days for employees 1 through 3:
10 20 5
The revised number of vacation days are:  .
Employee number 1 vacation days = 15
Employee number 2 vacation days = 25
Employee number 3 vacation days = 10
```

Упражнения для самопроверки

11. Рассмотрим такое определение функции:

```
void tripler(int& n)
{
    n = 3*n;
}
```

Какие из следующих строк:

```
int a[3] = {4, 5, 6}, number = 2;
tripler(number);
tripler(a[2]);
tripler(a[3]);
tripler(a[number]);
tripler(a);
```

являются допустимыми вызовами этой функции?

12. Что неверно (если что-либо неверно) в следующем фрагменте кода:

```
int b[5] = {1, 2, 3, 4, 5};
for (int i = 1; i <= 5; i++)
    tripler(b[i]);
```

Определение функции `tripler` приведено в предыдущем упражнении.

Массивы в качестве аргументов функций

Формальным параметром функции может быть целый массив. Такой параметр не является ни передаваемым по ссылке, ни передаваемым по значению, он относится к особому типу, называемому *параметром типа массива.*

В качестве примера рассмотрим программу листинга 10.3, где приведено определение функции с одним параметром типа массива, названным `a`. При вызове функции этот параметр заменяется целым массивом. Кроме того, у функции имеется один обыкновенный параметр (`size`), передаваемый по значению, в котором задается размер массива в виде целого числа. Функция заполняет передаваемый ей в качестве аргумента массив (его элементы) значениями, веденными с клавиатуры, а затем выводит на экран сообщение с указанием индекса последнего элемента массива.

Листинг 10.3. Функция с параметром типа массива

Объявление функции

```
void fill_up(int a[], int size);
// Предусловие: size – это объявленный размер массива a.
// Пользователь введет size целых чисел.
// Постусловие: массив a заполнен целыми числами
// в количестве size, веденными с клавиатуры.
```

Определение функции

```
// Используем библиотеку классов iostream.
void fill_up(int a[], int size)
{
    using namespace std;
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    size--;
    cout << "The last array index used is " << size << endl;
}
```

Формальный параметр `int a[]` является параметром типа массива, на что указывают пустые квадратные скобки. В большинстве случаев параметр такого типа ведет себя как параметр, передаваемый по ссылке, но все же эти два типа передачи параметров имеют определенные различия. Давайте подробно рассмотрим процесс передачи аргументов типа массива в нашем примере. (*Аргумент типа массива* — это обычный массив, который подставляется в параметр типа массива, такой как `a[]`.)

В вызове функции `fill_up` должно быть задано два аргумента: массив целых чисел и его объявленный размер. Вот, например, один из возможных вызовов этой функции:

```
int score[5], number_of_scores = 5;
fill_up(score, number_of_scores);
```

Такой вызов функции `fill_up` заполняет массив `score` пятью целыми числами, введенными с клавиатуры. Обратите внимание, что в записи формального параметра `a[]`, заданного в объявлении функции и в заголовке ее определения, используются квадратные скобки, но без индексного выражения. (В них можно задать индекс, однако компилятор его просто проигнорирует.) В то же время аргумент в вызове функции (в данном примере, `score`) задается без квадратных скобок и индексного выражения.

Что же происходит с аргументом типа массива `score` в этом вызове функции? Он подставляется в формальный параметр `a` в теле функции, после чего оно выполняется. Таким образом, вызов

```
fill_up(score, number_of_scores);
```

эквивалентен следующему коду:

```
{
    using namespace std;
    size = 5; // 5 – это значение параметра number_of_scores.
```

```
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> score[i];
    size--;
    cout << "The last array index used is "
         << size << endl;
}
```

Однако несмотря на внешнее сходство с параметрами, передаваемыми по ссылке, формальный параметр a отличается от тех параметров, с которыми мы имели дело до сих пор. Сначала выясним, в чем они сходны. Формальный параметр a отмечает место подстановки аргумента score. В случае вызова функции fill_up с аргументом типа массива score компьютер ведет себя так, словно параметр a заменен аргументом score. Когда в качестве аргумента в вызове функции задается массив, все действия, относящиеся к соответствующему параметру, выполняются непосредственно над аргументом, так что значения элементов массива могут изменяться функцией. Если в теле функции запрограммировано изменение формального параметра (например, с помощью оператора cin), в результате изменяется соответствующий аргумент, то есть массив в основной программе.

Описанное выше поведение параметра типа массива почти не отличается от поведения параметра, передаваемого по ссылке. Однако параметр-массив все же ведет себя несколько иначе. Для того чтобы понять это различие, нужно выяснить некоторые особенности массивов.

Напомним, что массив хранится в памяти в виде непрерывного блока. В качестве примера рассмотрим следующее объявление массива score:

```
int score[5];
```

Обнаружив в тексте программы такое объявление, компилятор резервирует память для пяти переменных типа int, расположенных в памяти друг за другом. При этом запоминается только адрес первого элемента score[0], а не всех элементов массива. Когда программа обращается к одному из элементов массива, например к score[3], компьютер вычисляет его адрес на основании адреса элемента score[0]. Он знает, что элемент score[3] располагается в памяти через три переменные типа int от начала score[0]. Поэтому для получения адреса элемента score[3] компьютер берет адрес элемента score[0] и прибавляет к нему число, представляющее количество памяти, занимаемой тремя переменными типа int; результат и является адресом элемента score[3].

Рассматриваемый с этой точки зрения массив характеризуется тремя компонентами: адресом (местоположением в памяти) первого элемента, базовым типом массива (определяющим, сколько памяти занимает каждый его элемент) и размером массива (то есть количеством его элементов). При использовании массива в качестве аргумента этой функции передается только первый из трех компонентов. Когда аргумент типа массива подставляется в соответствующий формальный параметр, параметру присваивается только адрес первого элемента массива. Базовый тип массива, передаваемого в качестве аргумента, должен соответствовать базовому типу формального параметра, поэтому функции известен также базовый тип массива. Однако в аргументе типа массива функции не передается никакая

информация о размере массива. При выполнении функции компьютер знает, где начинается массив и сколько места в памяти занимает каждый его элемент, но не имеет представления о количестве элементов. Вот почему вместе с массивом важно всегда передавать функции аргумент типа int, сообщающий размер массива. И именно в этом состоит отличие параметра типа массива от параметра, передаваемого по ссылке. Параметр типа массива можно считать особой формой параметра, передаваемого по ссылке, о котором функции сообщается все, кроме размера массива.

Параметры типа массива могут показаться вам немного необычными, но именно поэтому они обладают одним очень полезным свойством. Чтобы разобраться, в чем оно заключается, вновь обратимся к листингу 10.3. Функция fill_up может использоваться для заполнения массива любого размера — лишь бы его базовым типом был int. Предположим, у нас имеется следующее объявление массивов:

```
int score[5], time[10];
```

Тогда первый из приведенных ниже вызовов функции fill_up заполняет массив score пятью значениями, а второй заполняет массив time десятью значениями.

```
fill_up(score, 5);
fill_up(time, 10);
```

Как видите, одну и ту же функцию можно использовать для обработки массивов разного размера, поскольку размер массива задается при помощи отдельного аргумента, а не фиксируется жестко в определении параметра типа массива.

Квалификатор параметра const

Когда в вызове функции используется аргумент типа массива, функция может изменять хранящиеся в нем значения. Это конечно же, хорошо, но как быть, если функция вообще не должна модифицировать полученный массив? В достаточно сложной функции существует вероятность непреднамеренно запрограммировать изменение элементов этого массива. Для переменных, передаваемых функциям, эту проблему можно решить путем передачи аргумента по значению, а не по ссылке. Однако и для массива могут быть приняты меры предосторожности — следует сообщить компилятору, что функция не должна изменять данные полученного ею массива. Для этого в заголовке и в объявлении функции перед соответствующим формальным параметром, обозначающим массив, нужно поставить квалификатор const. Параметр типа массива, объявленный с таким квалификатором, называется *константным*.

В качестве примера рассмотрим следующую функцию, которая выводит значения из массива на экран, но ничего не изменяет в самом массиве:

```
void show_the_world(int a[], int size_of_a)
// Предусловие: size_of_a — это объявленный размер массива a.
// Всем элементам массива a присвоены значения.
// Постусловие: значения из массива a выведены на экран.
{
    cout << "The array contains the following values:\n";
```

```
    for (int i = 0; i < size_of_a; i++)
        cout << a[i] << " ";
    cout << endl;
}
```

Функция show_the_world будет работать правильно. Однако в качестве дополнительной меры предосторожности можно добавить в ее заголовок квалификатор const:

```
void show_the_world(const int a[], int size_of_a)
```

При наличии этого квалификатора компьютер будет следить за тем, чтобы функция не изменяла значение соответствующего параметра. И если в программе обнаружится попытка подобного изменения, компьютер выдаст сообщение об ошибке. Следующая версия функции show_the_world содержит ошибку — она изменяет значение аргумента типа массива. (Однако учтите, что при наличии квалификатора const, компилятор обнаружит эту ошибку и выведет соответствующее сообщение.)

```
void show_the_world(const int a[], int size_of_a)
// Предусловие: size_of_a – это объявленный размер массива a.
// Всем элементам массива a присвоены значения.
// Постусловие: значения из массива a выведены на экран.
{
    cout << "The array contains the following values:\n";
    for (int i = 0; i < size_of_a; a[i]++) // Ошибка, но при отсутствии квалификатора const
                                           // компилятор ее не заметит.
        cout << a[i] << " ";
    cout << endl;
}
```

Если же эта функция содержит указанную ошибку, но в ее определении не использован квалификатор const, то она успешно откомпилируется и запустится без вывода сообщения об ошибке. Однако цикл при этом будет выполняться бесконечно, увеличивая значение элемента a[0] массива и выводя на экран очередное его значение на каждой итерации.

Ошибка в этой версии функции show_the_world заключается в том, что в цикле for выполняется приращение не той величины. Вместо индекса i увеличивается значение элемента массива a[i]. В начале цикла переменной i присваивается значение 0, которое так никогда и не изменяется. Оператор a[i]++ должен увеличивать значение элемента массива. Но поскольку этот массив определен с квалификатором const, будет выведено сообщение об ошибке, указывающее, в каком месте программы она происходит.

Обычно помимо самого определения функции программа содержит и ее объявление. Поэтому если в заголовке определения функции задан квалификатор const, он должен быть задан и в объявлении функции.

Этот квалификатор можно использовать с любыми параметрами, но, как правило, он применяется только с параметрами типа массива и передаваемыми по ссылке параметрами типа класса (см. главу 8).

Формальные параметры и аргументы типа массива

Аргументом функции может быть целый массив, и тогда этот аргумент передается особым образом: не по ссылке и не по значению. Такие аргументы называются *аргументами типа массива*. Когда аргумент типа массива подставляется в параметр типа массива, функция получает единственное значение — адрес первого элемента массива в памяти (то есть элемента массива с индексом 0). Размер массива не передается функции автоматически, поэтому в ее объявлении обычно определяют дополнительный параметр типа int, в котором явно задается размер массива (как в приведенном ниже примере).

Аргумент типа массива очень похож на аргумент, передаваемый по ссылке: если в теле функции изменяется соответствующий параметр, это изменение на самом деле касается аргумента. Таким образом, функция может изменять значение элемента массива, который передан ей в качестве аргумента.

Синтаксис

возвращаемый_тип имя_функции (... , *базовый_тип_имя_массива*[], ...);

Пример

```
void sum_array(double& sum, double a[], int size);
```

Ловушка: несогласованное использование квалификатора const

Квалификатор параметра const действует по принципу «все или ничего». Если он используется для параметра типа массива конкретного типа в одной из функций программы, то должен применяться и во всех остальных функциях программы, имеющих параметр того же типа и не изменяющих его значение. Это связано с часто встречающимися в программах на C++ вызовами одних функций другими. Рассмотрим приведенное ниже определение функции show_difference и объявление используемой в нем функции compute_average:

```
double compute_average(int a[], int number_used);
// Возвращает среднее арифметическое первых number_used
// элементов массива a. Массив a не изменяется.

void show_difference(const int a[], int number_used)
{
    double average = compute_average(a, number_used);
    cout << "Average of the " << number_used
         << " numbers = " << average << endl
         << "The numbers are:\n";
    for (int index = 0; index < number_used; index++)
        cout << a[index] << " differs from average by "
             << (a[index]-average) << endl;
}
```

Большинство компиляторов выведет для данного кода сообщение об ошибке. Функция compute_average не изменяет параметр a. Но когда компилятор анализирует определение функции show_difference, он обнаруживает, что функция compute_average может изменять значение своего параметра типа массива. Это происходит потому, что во время обработки определения функции show_difference компилятору

ничего не известно о функции compute_average, кроме ее объявления. В этом объявлении отсутствует квалификатор const, из чего следует, что для функции compute_average изменение параметра a не запрещено. Чтобы определение функции show_difference было откомпилировано без сообщений об ошибках, объявление функции compute_average должно быть таким:

```
double compute_average(const int a[], int number_used);
```

Функции, возвращающие массивы

Функция не может вернуть массив тем же способом, каким она возвращает обычные значения типа int или double. Однако возможность получения массива в качестве результирующего значения функции все же существует: нужно получить от функции указатель на массив. Но вы пока не можете написать функцию, возвращающую указатель на массив, поскольку мы еще не рассматривали указатели. К этому вопросу мы вернемся в главе 12.

Пример: диаграмма производительности

В данном примере мы используем массивы при нисходящем проектировании программы. В качестве аргументов функции, реализующей подзадачи проекта, будут использоваться как элементы массивов, так и целые массивы.

Постановка задачи

Компания Apex Plastic Spoon Manufacturing поручила нам написать программу для вывода гистограммы объемов продукции, выпущенной четырьмя принадлежащими компании заводами за заданную неделю. Для каждой категории продукции каждый из заводов предоставляет отдельный количественный показатель (число чайных, столовых, обычных десертных, цветных десертных ложек и т. д.). При этом каждый завод может выпускать разное количество видов продукции и, соответственно, иметь различное количество цехов. Например, один завод производит только цветные ложечки для коктейля. Данные вводятся по очереди для каждого из заводов и включают последовательность чисел, представляющих объем продукции, выпущенной в каждом из цехов. Программа должна выводить гистограмму приблизительно такого вида:

```
Plant #1 **********
Plant #2 ************
Plant #3 ******************
Plant #4 *****
```

Каждая звездочка в гистограмме обозначает 1000 единиц продукции.

Мы решили считывать входные данные отдельно для каждого цеха завода. Поскольку цеха не могут выпускать отрицательное количество продукции, входное значение для каждого из цехов будет неотрицательным, поэтому отрицательное число можно использовать как сигнальное значение, указывающее на окончание ввода данных по очередному заводу.

Поскольку единицей измерения данных в результирующей гистограмме является 1000 ложек (либо другой производимой продукции), каждое полученное в результате вычислений значение нужно будет разделить на 1000. При этом следует учитывать, что компьютер может выводить только целое количество звездочек (не может же он вывести 1,6 звездочки для 1600 ложек), так что мы будем округлять результат до ближайшего целого числа тысяч. Например, значение 1600 будет округлено до 2000 и представлено в гистограмме двумя звездочками. Ниже приведено точное описание входных и выходных данных программы.

Входные данные. Имеются четыре завода, пронумерованных от 1 до 4. Для каждого из них вводится следующая информация: последовательность чисел, каждое из которых представляет собой объем продукции, выпущенной каждым цехом завода. Эта последовательность завершается отрицательным числом, служащим сигнальным значением окончания ввода.

Выходные данные. Гистограмма, представляющая общий объем продукции, выпущенной каждым заводом. Одна звездочка в гистограмме соответствует 1000 единиц продукции. Значение количества продукции, выпущенной каждым из заводов, округляется до ближайшей 1000 единиц.

Анализ задачи

Для хранения общего количества продукции, выпущенной каждым из заводов, будет использоваться массив, названный `production`. В C++ индексы массивов всегда начинаются с 0. А поскольку заводы нумеруются от 1 до 4, а не от 0 до 3, номер завода не может служить индексом для этого массива. То есть объем продукции завода номер *n* будет храниться в элементе массива `production[n-1]`. Например, объем продукции завода номер 1 будет храниться в элементе `production[0]`, объем продукции завода номер 2 — в элементе `production[1]` и т. д.

Поскольку результаты необходимо выводить в тысячах единиц, программа должна масштабировать значения элементов массива. Так, если завод номер 3 выпустил 4040 единиц продукции, элементу `production[2]` первоначально будет присвоено значение `4040`. Затем оно будет масштабировано до 4, в результате чего этому элементу будет присвоено значение 4, и в гистограмме для завода номер 3 будет выведено четыре звездочки.

Задачу программы можно разделить на следующие подзадачи.

- Ввод данных (`input_data`) — чтение входных данных для каждого завода и установка значения каждого элемента массива `production[plant_number-1]` (где `plant_number` — номер завода) равным общему количеству продукции, выпущенной данным заводом.
- Масштабирование (`scale`) — для каждого номера завода (`plant_number`) запись в элемент массива `production[plant_number-1]` количества звездочек, соответствующего объему выпущенной заводом продукции.
- Гистограмма (`graph`) — вывод гистограммы.

Функциям, выполняющим эти три подзадачи, будет передаваться в качестве аргумента весь массив `production`. Как обычно при использовании параметра типа массива вместе с ним должен быть определен еще один формальный параметр,

чтобы передать в функцию размер массива (в данном случае количество заводов). Количество заводов будет задаваться в виде именованной константы, и она же будет определять размер массива production. Главная часть программы, а также объявления функций, выполняющих ее подзадачи, и определение константы, представляющей собой количество заводов, приведены в листинге 10.4. Обратите внимание, что поскольку функции graph незачем изменять свой параметр типа массива, он объявлен как константный с помощью квалификатора const. Программный код листинга 10.4 является каркасом нашей программы. Сохранив его в отдельном файле, можно откомпилировать этот файл для выявления синтаксических ошибок до того, как будут написаны сами функции, реализующие подзадачи.

Теперь можно приступать к реализации функций. Мы поочередно разработаем алгоритм каждой из трех функций, напишем ее программный код и протестируем его.

Листинг 10.4. Каркас программы для построения гистограммы

```
// Считывает данные и выводит гистограмму объема продукции,
// произведенной четырьмя заводами.
#include <iostream>
const int NUMBER_OF_PLANTS = 4;

void input_data(int a[], int last_plant_number);
// Предусловие: last_plant_number – это объявленный размер массива a.
// Постусловие: элементы массива от plant_number = 1 до last_plant_number:
// a[plant_number-1] содержат значения общего объема продукции,
// выпущенной заводом номер plant_number.

void scale(int a[], int size);
// Предусловие: элементы массива от a[0] до a[size-1] содержат
// неотрицательные значения.
// Постусловие: каждому элементу массива a[i] присвоено количество
// тысяч единиц продукции, округленное до целого числа,
// для всех i, соответствующих условию 0 <= i <= size-1.

void graph(const int asterisk_count[], int last_plant_number);
// Предусловие: элементы массива от asterisk_count[0] до
// asterisk_count[last_plant_number-1] содержат
// неотрицательные значения.
// Постусловие: на экран выведена гистограмма, показывающая,
// что завод номер N выпустил asterisk_count[N-1] тысяч
// единиц продукции, для каждого N, соответствующего
// условию 1 <= N <= last_plant_number.

int main()
{
    using namespace std;
    int production[NUMBER_OF_PLANTS];

    cout << "This program displays a graph showing\n"
         << "production for each plant in the company.\n";
    input_data(production, NUMBER_OF_PLANTS);
    scale(production, NUMBER_OF_PLANTS);
    graph(production, NUMBER_OF_PLANTS);

    return 0;
}
```

Разработка алгоритма функции input_data

Объявление функции input_data и пояснительный комментарий к ней приведены в листинге 10.4. Если вы посмотрите на код функции main (показанной там же), то заметите, что при вызове функции input_data формальный параметр типа массива a заменяется аргументом типа массива production, и поскольку номер последнего завода равен количеству заводов, в формальный параметр last_plant_number подставляется константа NUMBER_OF_PLANTS. Алгоритм функции input_data очень прост.

Для каждого номера завода (plant_number) от 1 до last_plant_number нужно выполнить следующее:

- прочитать все данные для завода номер plant_number;
- сложить все числа;
- присвоить сумму элементу production[plant_number-1] массива production.

Программный код функции input_data

Алгоритм функции input_data реализуется с помощью приведенного ниже программного кода.

```
// Используем библиотеку классов iostream.
void input_data(int a[], int last_plant_number)
{
    using namespace std;
    for (int plant_number = 1;
            plant_number <= last_plant_number; plant_number++)
    {
        cout << endl
             << "Enter production data for plant number "
             << plant_number << endl;
        get_total(a[plant_number-1]);
    }
}
```

Поскольку основная работа по вводу данных и вычислению итоговых значений выполняется функцией get_total, код приведенной функции input_data несложный. Но прежде чем перейти к рассмотрению функции get_total, нужно кое-что сказать о функции input_data. Обратите внимание, что итоговое значение для завода номер plant_number записывается в элемент массива с индексом plant_number-1; как вы помните, массивы всегда индексируются с нуля, тогда как заводы нумеруются с единицы.

Ну а теперь поговорим о функции get_total. Она выполняет основную работу по вводу значений объемов выпуска продукции для очередного завода, складывает их и сохраняет сумму в элементе массива, соответствующем этому заводу. Однако функции get_total не обязательно знать, что ее аргумент является элементом массива. Она просто получает некоторую переменную типа int, переданную по ссылке. Это означает, что get_total — обычная функция для ввода данных, ничем не отличающаяся от других таких же функций, которыми мы пользовались еще до знакомства с массивами. Она считывает с клавиатуры последовательность чисел, оканчивающуюся сигнальным значением (отрицательным числом), складывает их

по мере чтения и присваивает итоговую сумму своему аргументу — переменной типа int. Как видите, в ней нет ничего нового. В листинге 10.5 приведены определения функций get_total и input_data. Они включены в простую отладочную программу.

Тестирование функции input_data

Каждая функция должна тестироваться в программе, в которой она является единственной непротестированной функцией. Наша функция input_data содержит вызов функции get_total, поэтому сначала в отдельной отладочной программе нужно протестировать функцию get_total. Только после этого ее можно будет использовать для тестирования функции input_data в программе, подобной приведенной в листинге 10.5.

В ходе тестирования функции input_data следует проверить все возможные типы входных значений для завода. В частности, необходимо выяснить, как она будет себя вести, если завод не произвел никакой продукции (как завод 4 в приведенном примере диалога), произвел только один вид продукции (как завод 3) и произвел несколько видов продукции (как заводы 1 и 2). Нужно проверить как нулевые, так и ненулевые входные значения. Поэтому в этом же примере диалога для завода 2 в последовательность вводимых данных включено значение 0.

Листинг 10.5. Программа для тестирования функции input_data

```
// Программа для отладки функции input_data.
#include <iostream>
const int NUMBER_OF_PLANTS = 4;

void input_data(int a[], int last_plant_number);
// Предусловие: last_plant_number - это объявленный размер массива a.
// Постусловие: элементы от plant_number = 1 до last_plant_number:
// a[plant_number-1] содержат значения общего объема продукции,
// выпущенной заводом номер plant_number.

void get_total(int& sum);
// Считывает с клавиатуры неотрицательные целые числа
// и помещает их сумму в параметр sum.

int main()
{
    using namespace std;
    int production[NUMBER_OF_PLANTS];
    char ans;

    do
    {
        input_data(production, NUMBER_OF_PLANTS);
        cout << endl
             << "Total production for each"
             << " of plants 1 through 4:\n";
        for (int number = 1;
                number <= NUMBER_OF_PLANTS; number++)
```

продолжение

Листинг 10.5 (*продолжение*)

```
        cout << production[number-1] << " ";

        cout << endl
             << "Test Again?(Type y or n and Enter): ";
        cin >> ans;
    } while ( (ans != 'N') && (ans != 'n') );

    cout << endl;

    return 0;
}

// Используем библиотеку классов iostream.
void input_data(int a[], int last_plant_number)
{
    using namespace std;
    for (int plant_number = 1;
             plant_number <= last_plant_number; plant_number++)
    {
        cout << endl
             << "Enter production data for plant number "
             << plant_number << endl;
        get_total(a[plant_number-1]);
    }
}

// Используем библиотеку клоссов iostream.
void get_total(int& sum)
{
    using namespace std;
    cout << "Enter number of units produced by each department.\n"
         << "Append a negative number to the end of the list.\n";

    sum = 0;
    int next;
    cin >> next;
    while (next >= 0)
    {
        sum = sum+next;
        cin >> next;
    }

    cout << "Total = " << sum << endl;
}
```

Пример диалога

```
Enter production data for plant number 1
Enter number of units produced by each department.
Append a negative number to the end of the list.
1 2 3 -1
Total = 6

Enter production data for plant number 2
Enter number of units produced by each department.
```

```
Append a negative number to the end of the list.
0 2 3 -1
Total = 5

Enter production data for plant number 3
Enter number of units produced by each department.
Append a negative number to the end of the list.
2 -1
Total = 2

Enter production data for plant number 4
Enter number of units produced by each department.
Append a negative number to the end of the list.
-1
Total = 0

Total production for each of plants 1 through 4:
6 5 2 0
Test Again?(Type y or n and Enter): n
```

Разработка алгоритма для функции scale

Функция scale изменяет значение каждого элемента массива production, присваивая ему количество звездочек, которое нужно вывести для соответствующего завода. Поскольку одна звездочка соответствует 1000 единиц продукции, первоначальное значение каждого элемента массива production должно быть разделено на 1000.0. Чтобы получить целое количества звездочек, результат необходимо округлить до ближайшего целого числа. Этот метод можно применять при масштабировании значений массива любого размера, поэтому объявление функции scale из листинга 10.4, которое повторно приведено ниже, выполнено с использованием терминов массива произвольного размера:

```
void scale(int a[], int size);
// Предусловие: элементы массива от a[0] до a[size-1] содержат
// неотрицательные значения.
// Постусловие: каждому элементу массива a[i] присвоено количество
// тысяч единиц продукции, округленное до целого числа,
// для всех i, соответствующих условию 0 <= i <= size-1.
```

При вызове функции scale вместо параметра типа массива a подставляется массив production, а вместо формального параметра size — константа NUMBER_OF_PLANTS, так что вызов функции выглядит следующим образом:

```
scale(production, NUMBER_OF_PLANTS);
```

Псевдокод алгоритма работы функции scale таков:

```
for (int index = 0; index < size; index++)

Разделить значение a[index] на 1000 и округлить результат до ближайшего целого числа;
результат округления сделать новым значением элемента массива a[index].
```

Программный код функции scale

Ниже алгоритм функции scale представлен в коде, написанном на C++. Из этой функции вызывается еще одна функция — round, которую нам только предстоит определить. Она принимает один аргумент типа double и возвращает значение

типа int — ближайшее к значению аргумента целое число; таким образом, функция round округляет значение своего аргумента до ближайшего целого.

```
void scale(int a[], int size)
{
    for (int index = 0; index < size; index++)
        a[index] = round(a[index]/1000.0);
}
```

Обратите внимание, что деление выполняется на 1000.0, а не на 1000 (без десятичной точки). Если бы мы делили исходное значение на 1000, C++ выполнял бы целочисленное деление. Например, 2600/1000 возвращает 2, тогда как 2600/1000.0 возвращает 2.6. После округления нам действительно нужно целое число, но мы хотим, чтобы для исходного значения 2600 получился результат 3, а не 2, то есть нам требуется округление результата деления, а не просто отсечение его дробной части.

Теперь обратимся к определению функции round, округляющей значение своего аргумента до ближайшего целого числа. Так, round(2.3) возвращает 2, а round(2.6) возвращает 3. Код функций round и scale приведен в листинге 10.6. Он требует некоторого пояснения.

В функции round используется предопределенная функция floor из библиотеки с заголовочным файлом cmath. Эта функция возвращает ближайшее целое число, которое меньше ее аргумента. То есть и floor(2.1), и floor(2.9) возвращает 2. Чтобы убедиться, что функция round правильно выполняет свою задачу, разберем несколько примеров. Рассмотрим вызов round(2.4), который возвращает значение

```
floor(2.4+0.5)
```

равное floor(2.9), то есть 2.0. Фактически для любого числа, большего или равного 2.0 и меньшего 2.5, функция floor возвращает 2.0, поскольку любое такое число плюс 0.5 меньше, чем 3.0. Поэтому для любого числа, большего или равного 2.0 и меньшего 2.5, функция round возвращает 2 (так как в ее объявлении указано, что она возвращает значение типа int, результат выполненных в ней вычислений — в нашем примере значение 2.0 — с помощью оператора приведения типа static_cast<int> преобразуется в 2).

Теперь рассмотрим числа, большие или равные 2.5; например, число 2.6. Вызов round(2.6) возвращает

```
floor(2.6+0.5)
```

значение floor(3.1), то есть 3.0. Фактически для любого числа, большего или равного 2.5 и меньшего или равного 3.0, функция floor возвращает 3.0, поскольку любое такое число плюс 0.5 больше, чем 3.0. Поэтому для любого числа, большего или равного 2.5 и меньшего или равного 3.0, функция round возвращает 3.

Таким образом, функция round правильно работает для всех аргументов в диапазоне между 2.0 и 3.0. Очевидно, что в этом наборе значений нет ничего особенного. Та же аргументация применима и к любым другим неотрицательным числам. Так что можно сказать, что функция round правильно работает для любых неотрицательных аргументов.

Тестирование функции scale

В листинге 10.6 приведена демонстрационная программа для функции scale, но реальная отладочная программа для функций round и scale будет несколько сложнее. В частности, она должна позволять тестировать эти функции многократно. Полные отладочные программы мы не приводим. При их создании учтите, что сначала должна быть полностью протестирована функция round (с помощью отдельной отладочной программы), а затем функция scale – с использованием отлаженной функции round. Программа для тестирования функции round должна проверить аргумент, равный 0, затем аргументы, округляемые до большего числа (например, 2.6), и аргументы, округляемые до меньшего числа (например, 2.3). Программа для тестирования функции scale должна проверять аналогичный набор значений, которые могут содержаться в массиве.

Листинг 10.6. Программа для тестирования функции scale

```
// Демонстрационная программа для тестирования функции scale.
#include <iostream>
#include <cmath>

void scale(int a[], int size);
// Предусловие: элементы массива от a[0] до a[size-1] содержат
// неотрицательные значения.
// Постусловие: каждому элементу массива a[i] присвоено количество
// тысяч единиц продукции, округленное до целого числа,
// для всех i, соответствующих условию 0 <= i <= size-1.


int round(double number);
// Предусловие: number >= 0.
// Возвращает значение параметра number,
// округленное до ближайшего целого.

int main()
{
    using namespace std;
    int some_array[4], index;

    cout << "Enter 4 numbers to scale: ";
    for (index = 0; index < 4; index++)
        cin >> some_array[index];

    scale(some_array, 4);

    cout << "Values scaled to the number of 1000s are: ";
    for (index = 0; index < 4; index++)
        cout << some_array[index] << " ";
    cout << endl;

    return 0;
}

void scale(int a[], int size)
{
    for (int index = 0; index < size; index++)
        a[index] = round(a[index]/1000.0);
```

продолжение

Листинг 10.6 (*продолжение*)

```
}

// Используем библиотеку классов cmath.
int round(double number)
{
    using namespace std;
    return static_cast<int>(floor(number+0.5));
}
```

Пример диалога

```
Enter 4 numbers to scale: 2600 999 465 3501
Values scaled to the number of 1000s are: 3 1 0 4
```

Функция graph

Полная программа для вывода гистограммы приведена в листинге 10.7. Поскольку функция graph очень проста, мы не описываем в подробностях процесс ее разработки.

Листинг 10.7. Программа для вывода гистограммы

```
// Считывает данные и выводит гистограмму объема продукции,
// произведенной четырьмя заводами.
#include <iostream>
#include <cmath>
const int NUMBER_OF_PLANTS = 4;

void input_data(int a[], int last_plant_number);
// Предусловие: last_plant_number – это объявленный размер массива a.
// Постусловие: элементы от plant_number = 1 до last_plant_number:
// a[plant_number-1] содержат значения общего объема продукции,
// выпущенной заводом номер plant_number.

void scale(int a[], int size);
// Предусловие: элементы массива от a[0] до a[size-1] содержат
// неотрицательные значения.
// Постусловие: каждому элементу массива a[i] присвоено количество
// тысяч единиц продукции, округленное до целого числа,
// для всех i, соответствующих условию 0 <= i <= size-1.

void graph(const int asterisk_count[], int last_plant_number);
// Предусловие: элементы массива от asterisk_count[0] до
// asterisk_count[last_plant_number-1] содержат
// неотрицательные значения.
// Постусловие: на экран выведена гистограмма, показывающая,
// что завод номер N выпустил asterisk_count[N-1] тысяч
// единиц продукции, для каждого N, соответствующего
// условию 1 <= N <= last_plant_number.

void get_total(int& sum);
// Считывает введенные с клавиатуры неотрицательные целые числа
// и помещает значение их суммы в параметр sum.

int round(double number);
// Предусловие: number >= 0.
// Возвращает значение параметра number,
// округленное до ближайшего целого.
```

```
void print_asterisks(int n);
// Выводит на экран n звездочек.

int main()
{
    using namespace std;
    int production[NUMBER_OF_PLANTS];

    cout << "This program displays a graph showing\n"
         << "production for each plant in the company.\n";
    input_data(production, NUMBER_OF_PLANTS);
    scale(production, NUMBER_OF_PLANTS);
    graph(production, NUMBER_OF_PLANTS);
    return 0;
}

// Используем библиотеку классов iostream.
void input_data(int a[], int last_plant_number)

... Остальная часть определения функции input_data приведена в листинге 10.5. ...

// Использует библиотеку iostream.
void get_total(int& sum)

... Остальная часть определения функции get_total приведена в листинге 10.5. ...

void scale(int a[], int size)

... Остальная часть определения функции scale приведена в листинге 10.6. ...

// Использует библиотеку cmath.
int round(double number)

... Остальная часть определения функции round приведена в листинге 10.6. ...

// Используем библиотеку классов iostream.
void graph(const int asterisk_count[], int last_plant_number)
{
    using namespace std;
    cout << "\nUnits produced in thousands of units:\n";
    for (int plant_number = 1; plant_number <= last_plant_number; plant_number++)
    {
        cout << "Plant #" << plant_number << " ";
        print_asterisks(asterisk_count[plant_number-1]);
        cout << endl;
    }
}
// Используем библиотеку классов iostream.
void print_asterisks(int n)
{
    using namespace std;
    for (int count = 1; count <= n; count++)
        cout << "*";
}
```

Пример диалога

```
This program displays a graph showing
production for each plant in the company.
```

```
Enter production data for plant number 1
Enter number of units produced by each department.
Append a negative number to the end of the list.
2000 3000 1000 -1
Total = 6000

Enter production data for plant number 2
Enter number of units produced by each department.
Append a negative number to the end of the list.
2050 3002 1300 -1
Total = 6352

Enter production data for plant number 3
Enter number of units produced by each department.
Append a negative number to the end of the list.
5000 4020 500 4348 -1
Total = 13868

Enter production data for plant number 4
Enter number of units produced by each department.
Append a negative number to the end of the list.
2507 6050 1809 -1
Total = 10366

Units produced in thousands of units:
Plant #1 ******
Plant #2 ******
Plant #3 **************
Plant #4 **********
```

Упражнения для самопроверки

13. Напишите определение функции one_more, которая принимает в качестве аргумента массив с базовым типом int и увеличивает значение каждого элемента этого массива на единицу. Если нужны еще какие-либо формальные параметры, добавьте их.

14. Рассмотрим определение функции:

```
void too2(int a[], int how_many)
{
    for (int index = 0; index < how_many; index++)
    a[index] = 2;
}
```

Какие из следующих вызовов:

```
int my_array[29];
too2(my_array, 29);
too2(my_array, 10);
too2(my_array, 55);
"Hey too2. Please, come over here."
int your_array[100];
too2(your_array, 100);
too2(my_array[3], 29);
```

являются допустимыми?

15. Вставьте квалификатор const перед теми параметрами типа массива, которые можно определить как константные:

```
void output(double a[], int size);
// Предусловие: элементы массива от a[0]
// до a[size-1] содержат значения.
// Постусловие: выведены все значения от a[0] до a[size-1].

void drop_odd(int a[], int size);
// Предусловие: элементы массива от a[0]
// до a[size-1] содержат значения.
// Постусловие: все нечетные значения от a[0]
// до a[size-1] заменены нулями.
```

16. Напишите определение функции out_of_order, которая возвращает значение типа int и имеет формальный параметр типа массива с базовым типом double, а также параметр size типа int. Функция должна проверять, расположены ли значения в массиве по порядку, то есть согласно условию

```
a[0] <= a[1] <= a[2] <= ...
```

Если это не так, функция возвращает индекс первого элемента массива, нарушающего порядок. Если же указанное выше условие выполняется, функция возвращает -1. В качестве примера рассмотрим следующее объявление:

```
double a[10] = {1.2, 2.1, 3.3, 2.5, 4.5, 7.9, 5.4, 8.7, 9.9, 1.0};
```

В данном массиве элементы a[2] и a[3] — первая пара, расположенная не по порядку, а a[3] — первый нарушающий порядок элемент, поэтому функция out_of_order должна вернуть значение 3. Если бы массив был отсортирован правильно, она должна была бы вернуть -1.

10.3. Использование массивов

Никогда не полагайтесь на общее впечатление, друг мой,
сосредоточьте внимание на мелочах.

Артур Конан Дойл

В этом разделе рассказывается о работе с частично заполненными массивами, а также о сортировке и поиске данных в массивах. В нем не содержится новый материал по C++, а просто приводятся практические рекомендации, связанные с параметрами типа массива.

Частично заполненные массивы

Нередко при написании программы размер массива, который будет в ней использоваться, заранее неизвестен, или же программист предполагает сделать его зависимым от обрабатываемых данных и при каждом запуске программы получать другое значение. В подобных случаях проще всего объявить массив достаточно большого размера, чтобы удовлетворить максимально возможные потребности

программы. Тогда она сможет использовать любую его часть, большую или меньшую, в зависимости от текущих потребностей.

Однако частично заполненные массивы требуют дополнительного внимания со стороны программиста — программа должна следить за тем, какая часть массива применяется в каждый конкретный момент, и не обращаться к элементам массива, не входящим в нее. Пример такой программы приведен в листинге 10.8. Она считывает список очков, набранных игроками в гольф, и показывает, насколько каждое значение отличается от среднего. Эта программа может обрабатывать список, содержащий от одного до десяти значений. Значения хранятся в массиве score, состоящем из десяти элементов, но программа использует столько его элементов, сколько значений содержится во введенном списке. Количество используемых элементов массива хранится в переменной number_used. Значения представлены элементами от score[0] до score[number_used-1].

Программа работает так, словно переменная number_used содержит объявленный размер массива и этот массив используется целиком. Например, эта переменная передается всем функциям, в которых используется частично заполненный массив. А поскольку с ее помощью функция следит за тем, чтобы не обратиться по недопустимому индексу, в большинстве случаев отпадает необходимость в аргументе, определяющем настоящий объявленный размер массива. Таким образом эту переменную используют функции show_difference и compute_average. Однако функции fill_array нужно знать и объявленный размер массива, потому что она не считывает из него значения, а заполняет массив, и ей важно не выйти за его пределы.

Листинг 10.8. Частично заполненный массив

```
// Выводит разницу между каждым элементом списка очков
// игроков в гольф и средним арифметическим этих элементов.
#include <iostream>
const int MAX_NUMBER_SCORES = 10;

void fill_array(int a[], int size, int& number_used);
// Предусловие: size – это объявленный размер массива a.
// Постусловие: number_used присвоено количество
// значений в массиве a. Элементам массива от a[0] до a[number_used-1]
// присвоены неотрицательные целые числа, введенные с клавиатуры.

double compute_average(const int a[], int number_used);
// Предусловие: элементы массива от a[0] до a[number_used-1] содержат
// значения; number_used > 0.
// Возвращает среднее арифметическое чисел от a[0] до a[number_used-1].

void show_difference(const int a[], int number_used);
// Предусловие: первые number_used элементов массива a содержат значения.
// Постусловие: на экран выведены значения разности между каждым из первых
// number_used элементов массива a и их средним арифметическим.

int main()
{
    using namespace std;
```

```
    int score[MAX_NUMBER_SCORES], number_used;

    cout << "This program reads golf scores and shows\n"
         << "how much each differs from the average.\n";

    cout << "Enter golf scores:\n";
    fill_array(score, MAX_NUMBER_SCORES, number_used);
    show_difference(score, number_used);

    return 0;
}

// Используем библиотеку классов iostream.
void fill_array(int a[], int size, int& number_used)
{
    using namespace std;
    cout << "Enter up to " << size
         << " nonnegative whole numbers.\n"
         << "Mark the end of the list "
         << "with a negative number.\n";

    int next, index = 0;
    cin >> next;
    while ((next >= 0) && (index < size))
    {
        a[index] = next;
        index++;
        cin >> next;
    }

    number_used = index;
}

double compute_average(const int a[], int number_used)
{
    double total = 0;
    for (int index = 0; index < number_used; index++)
        total = total+a[index];
    if (number_used > 0)
    {
        return (total/number_used);
    }
    else
    {
        using namespace std;
        cout << "ERROR: number of elements "
             << "is 0 in compute_average.\n"
             << "compute_average returns 0.\n";
        return 0;
    }
}
void show_difference(const int a[], int number_used)
{
```

продолжение

Листинг 10.8 (*продолжение*)

```
    using namespace std;
    double average = compute_average(a, number_used);
    cout << "Average of the " << number_used
         << " scores = " << average << endl
         << "The scores are:\n";
    for (int index = 0; index < number_used; index++)
    cout << a[index] << " differs from average by "
         << (a[index]-average) << endl;
}
```

Пример диалога

```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
69 74 68 -1
Average of the 3 scores = 70.3333
The scores are:
69 differs from average by -1.33333
74 differs from average by 3.66667
68 differs from average by -2.33333
```

Совет программисту: не ограничивайте количество формальных параметров

Посмотрите на функцию `fill_array` в листинге 10.8. В качестве одного из аргументов ей передается объявленный размер массива `score` — `MAX_NUMBER_SCORES`, как в следующем вызове:

```
fill_array(score, MAX_NUMBER_SCORES, number_used);
```

`MAX_NUMBER_SCORES` — глобально определенная константа, и функция `fill_array` использует ее показанным выше способом. Возникает вопрос: зачем же передавать эту константу через параметр функции. Если бы функция `fill_array` применялась только в программе из листинга 10.8, она действительно могла пользоваться непосредственно константой `MAX_NUMBER_SCORES`. Однако это универсальная функция, с помощью которой можно заполнять любой массив, и ее можно применять в других программах. Например, она используется в программе из листинга 10.9, рассмотренной в следующем подразделе. Но там объявленный размер массива представлен другой именованной константой. Так что если бы функция `fill_array` работала непосредственно с константой `MAX_NUMBER_SCORES`, этой функцией нельзя было бы пользоваться в программе из листинга 10.9.

Но даже когда функция `fill_array` применяется в единственной программе, лучше передать ей объявленный размер массива в качестве аргумента. Таким образом значение `MAX_NUMBER_SCORES` выделяется как важная информация, необходимая для работы функции.

Пример: поиск в массиве

Одной из распространенных программных задач является поиск в массиве заданного значения. Например, массив может содержать номера, присвоенные студентам, слушающим некоторый курс. Для того чтобы узнать, является ли студент слушателем этого курса, нужно выяснить, содержится ли в массиве его номер. Программа из листинга 10.9 заполняет массив и затем ищет в нем заданные пользователем значения. Разумеется, реальные прикладные программы намного сложнее, наш же пример просто демонстрирует алгоритм последовательного просмотра массива. Алгоритм *последовательного просмотра* — это простейший из возможных алгоритмов поиска данных в массиве. Программа просматривает элементы массива по порядку от первого до последнего и сравнивает каждый из них с заданным значением.

Листинг 10.9. Поиск в массиве

```
// Поиск в частично заполненном массиве
// значений неотрицательных целых чисел.
#include <iostream>
const int DECLARED_SIZE = 20;

void fill_array(int a[], int size, int& number_used);
// Предусловие: size – это объявленный размер массива a.
// Постусловие: number_used присвоено количество
// значений в массиве a.
// Элементам массива от a[0] до a[number_used-1] присвоены
// неотрицательные целые числа, введенные с клавиатуры.

int search(const int a[], int number_used, int target);
// Предусловие: number_used <= объявленного размера массива a.
// Элементы массива от a[0] до a[number_used-1] содержат значения.
// Возвращает первый индекс, для которого a[index] == target,
// если таковой имеется; в противном случае возвращает -1.

int main()
{
    using namespace std;
    int arr[DECLARED_SIZE], list_size, target;

    fill_array(arr, DECLARED_SIZE, list_size);

    char ans;
    int result;
    do
    {
        cout << "Enter a number to search for: ";
        cin >> target;

        result = search(arr, list_size, target);
        if (result == -1)
            cout << target << " is not on the list.\n";
        else
            cout << target << " is stored in array position "
                 << result << endl
                 << "(Remember: The first position is 0.)\n";
```

продолжение

Листинг 10.9 (*продолжение*)

```
        cout << "Search again?(y/n followed by Enter): ";
        cin >> ans;
    } while ((ans != 'n') && (ans != 'N'));

    cout << "End of program.\n";
    return 0;
}

// Используем библиотеку классов iostream.
void fill_array(int a[], int size, int& number_used)

... Остальная часть определения функции fill_array приведена в листинге 10.8. ...

int search(const int a[], int number_used, int target)
{

    int index = 0;
    bool found = false;
    while ((!found) && (index < number_used))
        if (target == a[index])
            found = true;
        else
            index++;

    if (found)
        return index;
    else
        return -1;
}
```

Пример диалога

```
Enter up to 20 nonnegative whole numbers.
Mark the end of the list with a negative number.
10 20 30 40 50 60 70 80 -1
Enter a number to search for: 10
10 is stored in array position 0
(Remember: The first position is 0.)
Search again?(y/n followed by Enter): y
Enter a number to search for: 40
40 is stored in array position 3
(Remember: The first position is 0.)
Search again?(y/n followed by Enter): y
Enter a number to search for: 42
42 is not on the list.
Search again?(y/n followed by Enter): n
End of program.
```

В программе из листинга 10.9 поиск выполняет функция search. Задача поиска в массиве часто сложнее, чем просто нахождение заданного значения. Если это значение имеется в массиве, обычно требуется узнать его индекс, с помощью которого о нем можно получить некоторую дополнительную информацию. Поэтому наша функция search возвращает индекс заданного значения в массиве, а если оно не найдено, возвращает -1.

Рассмотрим эту функцию подробнее. В ней используется цикл while, с помощью которого элементы массива по очереди проверяются на равенство заданному значению. Переменная found используется как флаг, указывающий, найден ли элемент. Если он найден, переменная found получает значение true и цикл while завершается.

Пример: сортировка массива

Одной из самых распространенных и наиболее тщательно изученных задач программирования является сортировка набора значений, например сортировка объемов продаж какой-либо продукции в порядке убывания или сортировка слов по алфавиту. В этом разделе описывается функция sort, сортирующая частично заполненный массив чисел по возрастанию.

Эта функция имеет параметр a, в котором ей передается частично заполненный массив, и параметр number_used, в котором указывается количество заполненных элементов массива. Объявление и предусловие функции таковы:

```
void sort(int a[], int number_used);
// Предусловие: number_used <= объявленного
// размера массива a.
// Элементы массива от a[0] до a[number_used-1]
// содержат значения.
```

Функция sort сортирует элементы массива a следующим образом:

$a[0] \le a[1] \le a[2] \le \ldots \le a[number_used-1]$

Алгоритм, с помощью которого она выполняет свою задачу, называется *сортировкой методом выбора*. Это один из простейших и наиболее понятных алгоритмов сортировки.

Разработка алгоритма может быть основана на постановке задачи. В данном случае задача заключается в сортировке массива по возрастанию значений. Это означает, что значения нужно переупорядочить так, чтобы в a[0] хранилось наименьшее значение, в a[1] — наименьшее из оставшихся значений и т. д. Таким образом, сама постановка задачи определяет схему алгоритма сортировки методом выбора (поместить элемент с наименьшим индексом в a[index]):

```
for (int index = 0; index < number_used; index++)
```

Существует множество способов реализации этой схемы. Например, можно использовать два массива и копировать элементы из одного в другой в требуемом порядке, но можно обойтись и одним массивом, что экономнее и эффективнее. Поэтому в нашей функции sort для сортировки используется один единственный массив, тот самый, который она получает в качестве аргумента. В нем же хранятся сортируемые значения. Функция переупорядочивает значения в этом массиве, попарно меняя их местами. Рассмотрим алгоритм ее работы на конкретном примере, используя массив, показанный на рис. 10.2. Согласно алгоритму наименьшее значение помещается в элемент a[0] массива. Наименьшим является значение элемента a[3]. Поэтому меняются местами значения a[0] и a[3]. Затем среди оставшихся элементов a[1], a[2], a[3], ... , a[9] отыскивается следующее наименьшее

значение. В нашем примере это значение элемента a[5], которое меняется местами со значением элемента a[1]. Далее алгоритм находит и помещает на нужное место третий наименьший элемент, затем четвертый и т. д.

По мере сортировки все большая часть массива, начиная с начала, заполняется правильно отсортированными значениями. Обратите внимание, что в самом конце ничего не надо делать с последним элементом массива, a[9]. Если все остальные значения расположены правильно, a[9] тоже содержит верное значение. Ведь когда в неотсортированной части массива, откуда нужно выбрать наименьшее значение, остается только элемент a[9], он и является наименьшим значением, которое нужно поменять местами с самим собой.

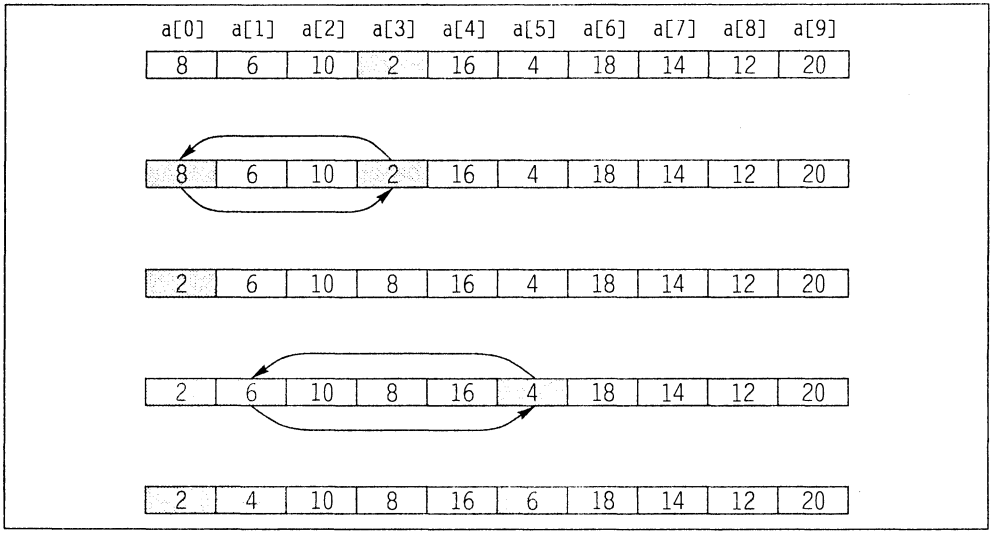


Рис. 10.2. Порядок сортировки

В листинге 10.10 приведено определение функции sort в составе демонстрационной программы. Для поиска наименьшего элемента в неотсортированной части массива эта функция вызывает функцию index_of_smallest, а затем перемещает найденный элемент в нужную позицию, меняя местами с тем элементом, который ее занимает. Последнее осуществляется с помощью функции swap_values (описанной в главе 4), также приведенной в листинге 10.10. Например, вызов

```
swap_values(a[0], a[3]);
```

меняет местами значения элементов a[0] и a[3].

Листинг 10.10. Сортировка массива

```
// Программа тестирования процедуры sort.
#include <iostream>

void fill_array(int a[], int size, int& number_used);
// Предусловие: size – это объявленный размер массива a.
// Постусловие: number_used присвоено количество значений в массиве a.
```

```
// Элементам массива от a[0] до a[number_used-1] присвоены
// неотрицательные целые числа, введенные с клавиатуры.

void sort(int a[], int number_used);
// Предусловие: number_used <= объявленного размера массива a.
// Элементы массива от a[0] до a[number_used-1] содержат значения.
// Постусловие: значения от a[0] до a[number_used-1]
// упорядочены таким образом,
// что a[0] <= a[1] <= ... <= a[number_used-1].

void swap_values(int& v1, int& v2);
// Меняет местами значения v1 и v2.

int index_of_smallest(const int a[],
                      int start_index, int number_used);
// Предусловие: 0 <= start_index < number_used.
// Элементы массива от a[start_index] до a[number_used-1]
// содержат значения.
// Возвращает индекс i элемента массива a[i],
// содержащего наименьшее из значений
// a[start_index], a[start_index+1], ..., a[number_used-1].

int main()
{
    using namespace std;
    cout << "This program sorts numbers "
         << "from lowest to highest.\n";

    int sample_array[10], number_used;
    fill_array(sample_array, 10, number_used);
    sort(sample_array, number_used);

    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < number_used; index++)
        cout << sample_array[index] << " ";
    cout << endl;

    return 0;
}

// Используем библиотеку классов iostream.
void fill_array(int a[], int size, int& number_used)

... Остальная часть определения функции fill_array приведена в листинге 10.9. ...

void sort(int a[], int number_used)
{
    int index_of_next_smallest;
    for (int index = 0; index < number_used-1; index++)
    {// Помещаем в a[index] правильное значение:
        index_of_next_smallest =
                index_of_smallest(a, index, number_used);
        swap_values(a[index], a[index_of_next_smallest]);
        // a[0] <= a[1] <=...<= a[index] - наименьшие и
        // уже отсортированные элементы исходного массива.
        // Остальные элементы пока находятся на своих позициях.
    }
}
```

продолжение

Листинг 10.10 (*продолжение*)

```
void swap_values(int& v1, int& v2)
{
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}


int index_of_smallest(const int a[],
                      int start_index, int number_used)
{
    int min = a[start_index],
        index_of_min = start_index;
    for (int index = start_index+1;
                      index < number_used; index++)
        if (a[index] < min)
        {
            min = a[index];
            index_of_min = index;
            // min – наименьший из элементов
            // от a[start_index] до a[index].
        }

    return index_of_min;
}
```

Пример диалога

```
This program sorts numbers from lowest to highest.
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
80 30 50 70 60 90 20 30 40 -1
In sorted order the numbers are:
20 30 30 40 50 60 70 80 90
```

Упражнения для самопроверки

17. Напишите программу, считывающую до десяти неотрицательных целых чисел в массив с именем number_array и выводящую их на экран. В данном упражнении можно обойтись без функций – это должна быть простейшая программа минимального размера.

18. Напишите программу, считывающую не более десяти букв в массив и выводящую их на экран в обратном порядке. Например, если пользователь ввел

    abcd.

    программа должна выводить:

    dcba

    В качестве сигнального значения, отмечающего конец ввода, используйте точку. Назовите массив letter_box. В данном упражнении можно обойтись без функций – это должна быть простейшая программа минимального размера.

19. Ниже приведено объявление альтернативной версии функции search, определенной в листинге 10.10.

```
bool search(const int a[], int number_used,
                           int target, int& where);
// Предусловие: number_used <= объявленного
// размера массива a.
// Элементы массива от a[0] до a[number_used-1] содержат значения.
// Постусловие: если target – один из элементов от a[0]
// до a[number_used-1], данная функция возвращает значение
// true и устанавливает значение параметра where
// таким образом, что a[where] == target;
// в противном случае функция возвращает false,
// и значение параметра where остается неизменным.
```

Для использования этой версии функции нужно немного модифицировать программу, но в данном упражнении вам следует только написать определение функции.

10.4. Массивы и классы

Комбинируя массивы, структуры и классы, можно создавать такие сложные типы, как массивы структур, массивы объектов и классы, членами которых являются массивы. В этом разделе рассматривается несколько простых примеров, демонстрирующих эти возможности.

Массивы структур и массивы классов

Базовым типом массива может быть любой тип данных, в том числе и тип, определенный программистом, например структура или класс. Если нужно, чтобы каждый элемент массива содержал набор значений разных типов, можно создать массив структур. Предположим, что нам требуется массив с десятью показателями погоды, в котором каждый показатель состоит из двух значений: скорости ветра и его направления (северного, южного, западного или восточного). Вот способ определения этого массива и структуры для его элементов:

```
struct WindInfo
{
    double velocity; // Скорость ветра в милях в час.
    char direction;  // 'N', 'S', 'E' или 'W'.
};

WindInfo data_point[10];
```

Чтобы заполнить массив `data_point`, можно написать следующий цикл `for`:

```
int i;
for (i = 0; i < 10; i++)
{
    cout << "Enter velocity for "
         << i << " numbered data point:";
    cin >> data_point[i].velocity;
    cout << "Enter direction for that data point"
         << " (N, S, E, or W): ";
    cin >> data_point[i].direction;
}
```

Выражения вроде data_point[i].velocity следует читать слева направо и очень внимательно. Здесь data_point — массив, а data_point[i] — элемент данного массива с индексом i, имеющий тип WindInfo и представляющий собой структуру с двумя переменными-членами: velocity и direction. Из этого следует, что data_point[i].velocity является переменной-членом с именем velocity i-го элемента массива data_point. Она хранит скорость ветра для i-го показателя погоды. А data_point[i].direction — это направление ветра для i-го показателя.

Десять элементов данных, хранящихся в массиве data_point, можно вывести на экран с помощью следующего цикла for:

```
for (i = 0; i < 10; i++)
    cout << "Wind data point number " << i << ": \n"
         << data_point[i].velocity
         << " miles per hour\n"
         << "direction " << data_point[i].direction
         << endl;
```

В листинге 10.11 показан файл интерфейса класса с именем Money. Объекты этого класса представляют денежные суммы в валюте США. Определения функций-членов, операторов-членов и дружественных функций класса Money приведены в листингах 8.3–8.7 и в ответе к упражнению 13 из главы 8. Эти определения можно объединить в файл реализации класса. Однако мы не приводим этот файл, поскольку для использования класса Money достаточно его интерфейса.

Класс Money может служить базовым типом массива. Простейший пример такого его использования приведен в листинге 10.12. Программа из этого листинга считывает список, содержащий пять денежных сумм, и вычисляет разность между каждым значением и максимальным из пяти значений. Обратите внимание, что мы оперируем массивом, базовым типом которого является класс, точно так же, как любым другим массивом. Фактически листинг программы 10.12 напоминает листинг программы 10.1 с той разницей, что в программе из листинга 10.12 базовым типом массива является класс.

Листинг 10.11. Файл интерфейса класса Money

```
// Это заголовочный файл money.h. Он содержит интерфейс класса Money.
// Значениями этого типа являются денежные суммы в валюте США.
#ifndef MONEY_H
#define MONEY_H
#include <iostream>
using namespace std;
namespace moneysavitch
{

  class Money
  {
  public:
     friend Money operator +(const Money& amount1, const Money& amount2);
     // Возвращает сумму значений amount1 и amount2.

     friend Money operator -(const Money& amount1, const Money& amount2);
     // Возвращает разность значений amount1 и amount2.

     friend Money operator -(const Money& amount);
     // Возвращает значение объекта amount, взятое с противоположным знаком.
```

```
    friend bool operator ==(const Money& amount1, const Money& amount2);
    // Возвращает true, если amount1 и amount2 содержат
    // одинаковые значения; в противном случае возвращает false.

    friend bool operator
          < (const Money& amount1, const Money& amount2);
    // Возвращает true, если amount1 меньше amount2,
    // и false в противном случае.

    Money(long dollars, int cents);
    // Инициализирует объект таким образом, чтобы он
    // содержал заданное в аргументах количество
    // долларов и центов. Если значение суммы отрицательно,
    // то и dollars, и cents должны быть отрицательными.

    Money(long dollars);
    // Инициализирует объект таким образом,
    // чтобы он содержал значение $dollars.00.

    Money();
    // Инициализирует объект таким образом,
    // чтобы он содержал значение $0.00.

    double get_value() const;
    // Возвращает значение денежной суммы,
    // записанной в вызывающем объекте.

    friend istream& operator >>(istream& ins, Money& amount);
    // Перегружает оператор >>,
    // чтобы им можно было пользоваться
    // для ввода значений типа Money.
    // Отрицательные суммы вводятся как -$100.00.
    // Предусловие: если ins – файловый входной поток,
    // то он уже подключен к файлу.

    friend ostream& operator
                  <<(ostream& outs, const Money& amount);
    // Перегружает оператор <<,
    // чтобы им можно было пользоваться
    // для вывода значений типа Money.
    // Перед каждым значением типа Money
    // выводит символ доллара.
    // Предусловие: если outs – файловый выходной поток,
    // то он уже подключен к файлу.
  private:
    long all_cents;
  };
} // Пространство имен moneysavitch.
#endif // MONEY_H
```

Массив объектов типа класса обрабатывается так же, как массив, базирующийся на простом типе (int или double). Например, разности между значениями, хранящимися в массиве difference, и максимальным из этих значений вычисляются так:

```
Money difference[5];
for (i = 0; i < 5; i++)
    difference[i] = max-amount[i];
```

Листинг 10.12. Программа, демонстрирующая работу с массивом объектов

```
// Считывает 5 денежных сумм и показывает, насколько
// каждая из них отличается от наибольшей суммы.
#include <iostream>
#include "money.h"

int main()
{
    using namespace std;
    using namespace moneysavitch;
    Money amount[5], max;
    int i;

    cout << "Enter 5 amounts of money:\n";
    cin >> amount[0];
    max = amount[0];
    for (i = 1; i < 5; i++)
    {
        cin >> amount[i];
        if (max < amount[i])
            max = amount[i];
        // max - это максимальное из значений
        // amount[0], ..., amount[i].
    }

    Money difference[5];
    for (i = 0; i < 5; i++)
        difference[i] = max-amount[i];

    cout << "The highest amount is " << max << endl;
    cout << "The amounts and their\n"
         << "differences from the largest are:\n";
    for (i = 0; i < 5; i++)
    {
        cout << amount[i] << " off by "
             << difference[i] << endl;
    }

    return 0;
}
```

Пример диалога

```
Enter 5 amounts of money:
$5.00 $10.00 $19.99 $20.00 $12.79
The highest amount is $20.00
The amounts and their
differences from the largest are:
$5.00 off by $15.00
$10.00 off by $10.00
$19.99 off by $0.01
$20.00 off by $0.00
$12.79 off by $7.21
```

Упражнения для самопроверки

20. Приведите определение типа для структуры Score с двумя переменными-членами — home_team и opponent типа int. Объявите массив с именем game из десяти элементов типа Score. Этот массив может использоваться для хранения очков каждой из десяти игр, в которых участвовала спортивная команда.
21. Напишите программу, считывающую пять денежных сумм, удваивающую каждую из них и выводящую результаты на экран. Воспользуйтесь массивом с базовым типом Money.

    Подсказка. В качестве образца можете использовать программу, приведенную в листинге 10.12, но ваша программа должна быть проще.

Массивы как члены классов

Членами структуры или класса могут быть переменные типа массива. Предположим, что спортсмену-пловцу нужна программа для учета заплывов на разные дистанции. В ней можно задать структуру my_best типа Data (определение которого приведено ниже) для хранения длины дистанции в метрах и времени каждого из выполненных спортсменом десяти заплывов на данную дистанцию:

```
struct Data
{
    double time[10];
    int distance;
};
Data my_best;
```

Структура my_best состоит из двух членов: переменной distance типа int, хранящей длину дистанции, и массива time из десяти элементов типа double для хранения времени каждого из десяти заплывов на данную дистанцию. Чтобы задать дистанцию длиной 20 метров, нужно выполнить следующий оператор:

```
my_best.distance = 20;
```

Массив можно заполнить значениями, введенными с клавиатуры:

```
cout << "Enter ten times (in seconds):\n";
for (int i = 0; i < 10; i++)
    cin >> my_best.time[i];
```

Выражение my_best.time[i] читается слева направо: my_best — структура; my_best.time — переменная-член этой структуры с именем time. Поскольку my_best.time — массив, для ссылки на его элемент добавляется индекс. Таким образом, my_best.time[i] — это i-й элемент массива my_best.time. Если в программе используется не структура, а класс, операции с массивом могут выполняться с помощью функций-членов данного класса, и тогда выражения для доступа к элементам массива упрощаются. Эту ситуацию иллюстрирует следующий пример.

Пример: класс для частично заполненного массива

В листингах программ 10.13 и 10.14 показано определение класса TemperatureList, объекты которого предназначены для хранения списков температур. Объект типа

TemperatureList может использоваться в программе, анализирующей погоду. Список температур хранится в переменной-члене list, определенной как массив. Поскольку этот массив обычно заполнен частично, для отслеживания количества внесенных в него элементов добавлена вторая переменная-член с именем size. Значением названной переменной является количество элементов массива list, применяемых для хранения значений.

В программе, использующей данный класс, заголовочный файл должен быть указан с помощью следующей директивы include:

```
#include "templist.h"
```

Объект типа TemperatureList объявляется подобно объекту любого другого типа. Например, оператор

```
TemperatureList my_data;
```

объявляет переменную my_data как объект типа TemperatureList.

Это объявление создает новый объект типа TemperatureList и вызывает для него используемый по умолчанию конструктор. В результате переменная-член size объекта my_data инициализируется значением 0, указывающим, что список пуст.

Объявив объект, можно с помощью функции-члена add_temperature добавить значение в перечень температур (то есть в массив-член list):

```
my_data.add_temperature(77);
```

Это единственный способ добавления значений в список объекта my_data, поскольку массив list является закрытым членом класса. Обратите внимание, что функция add_temperature сначала проверяет, остались ли в массиве незаполненные элементы, и добавляет новый элемент только в том случае, если для него есть место.

Листинг 10.13. Интерфейс класса с переменной-членом типа массива

```
// Это заголовочный файл templist.h.
// Он содержит интерфейс класса
// TemperatureList. Объекты этого класса хранят
// значения температуры по Фаренгейту.

#ifndef TEMPLIST_H
#define TEMPLIST_H
#include <iostream>
using namespace std;
namespace tlistsavitch
{
  const int MAX_LIST_SIZE = 50;

  class TemperatureList
  {
  public:
     TemperatureList();
     // Инициализирует объект пустым списком.

     void add_temperature(double temperature);
     // Предусловие: список не полон.
```

```
    // Постусловие: в список добавлено
    // заданное значение температуры.

    bool full() const;
    // Возвращает true, если список полон,
    // и false в противном случае.

    friend ostream& operator
          <<(ostream& outs, const TemperatureList& the_object);
    // Перегружает оператор <<,
    // чтобы им можно было пользоваться
    // для вывода значений типа TemperatureList.
    // Каждое значение выводится с новой строки.
    // Предусловие: если outs – файловый выходной поток,
    // он уже подключен к файлу.

  private:
    double list[MAX_LIST_SIZE]; // Значения температуры по Фаренгейту.
    int size; // Количество заполненных элементов массива.
  };
} // Пространство имен tlistsavitch.
#endif // TEMPLIST_H
```

Листинг 10.14. Реализация класса с членом-массивом

```
// Это файл реализации класса TemperatureList (templist.cpp).
// Интерфейс класса TemperatureList содержится в файле templist.h.
#include <iostream>
#include <cstdlib>
#include "templist.h"
using namespace std;
namespace tlistsavitch
{
    TemperatureList::TemperatureList() : size(0)
    {
        // Тело функции намеренно оставлено пустым.
    }

    void TemperatureList::add_temperature(double temperature)
    {// Использует библиотеки iostream и cstdlib.
        if ( full() )
        {
            cout << "Error: adding to a full list.\n";
            exit(1);
        }
        else
        {
            list[size] = temperature;
            size = size+1;
        }
    }
```

продолжение

Листинг 10.14 (*продолжение*)

```
bool TemperatureList::full() const
{
     return (size == MAX_LIST_SIZE);
}

// Используем библиотеку классов iostream.
ostream& operator
         <<(ostream& outs, const TemperatureList& the_object)
{
   for (int i = 0; i < the_object.size; i++)
        outs << the_object.list[i] << " F\n";
    return outs;
}
} // Пространство имен tlistsavitch.
```

Класс TemperatureList имеет очень ограниченную область применения. Над его объектами можно выполнять только следующие операции: инициализация списка как пустого, добавление в него элементов, проверка на заполнение списка и вывод содержащихся в нем значений. Для вывода значений температур, хранящихся в объявленном выше объекте my_data, нужно выполнить оператор

```
cout << my_data;
```

Класс TemperatureList не позволяет удалять значения из списка. Однако можно стереть его весь и начать заполнение сначала, вызвав используемый по умолчанию конструктор. Это делается так:

```
my_data = TemperatureList();
```

Класс TemperatureList не содержит никаких свойств, специфических именно для температур. Подобный класс можно определить для списка значений давления или расстояния либо любых других значений типа double. А чтобы в каждом из этих случаев не определять новый класс, можно создать единственный класс, представляющий произвольный список значений типа double независимо от того, для каких целей он будет использоваться. Такой класс вам предлагается определить в задании 10.

Упражнения для самопроверки

22. Модифицируйте класс TemperatureList, определенный в листингах 10.13 и 10.14, добавив в него функцию-член get_size, не имеющую аргументов и возвращающую количество значений температуры в списке.
23. Модифицируйте класс TemperatureList, определенный в листингах 10.13 и 10.14, добавив в него функцию-член get_temperature с одним аргументом типа int, значение которого больше или равно 0 и меньше MAX_LIST_SIZE. Функция возвращает значение типа double, равное значению заданного элемента списка. Для аргумента 0 она возвращает первое значение температуры, для аргумента 1 — второе и т. д. Предполагается, что функции не может быть передан индекс элемента, не содержащего значение температуры.

10.5. Многомерные массивы

Два индекса лучше, чем один.

Надпись на стене в комнате отдыха факультета вычислительной техники

Язык C++ позволяет объявлять массивы с несколькими индексами. О таких массивах рассказывается в этом разделе.

Основные понятия

Иногда полезно иметь массив с более чем одним индексом, и C++ это допускает. Следующий оператор:

```
char page[30][100];
```

объявляет массив символов с именем page, у которого два индекса: первый с диапазоном значений от 0 до 29 и второй с диапазоном значений от 0 до 99.

Все элементы этого массива имеют по два индекса, скажем, page[0][0], page[15][32], page[29][99]. Обратите внимание, что каждый индекс элемента должен быть заключен в отдельные квадратные скобки. Как и в случае одномерного массива, каждый элемент многомерного массива представляет собой переменную его базового типа.

Многомерный массив может иметь любое количество индексов, но чаще всего используют только два. Двухмерный массив можно представить в виде таблицы, при этом первый индекс будет определять строку, а второй — столбец. Например, объявленный выше двухмерный массив page для наглядности можно представить так:

```
page[0][0], page[0][1], ..., page[0][99]
page[1][0], page[1][1], ..., page[1][99]
page[2][0], page[2][1], ..., page[2][99]
.
.
.
page[29][0], page[29][1], ..., page[29][99]
```

Этот массив можно использовать для хранения символов текстовой страницы длиной в 30 строк (пронумерованных от 0 до 29) по 100 символов в строке (пронумерованных от 0 до 99).

В C++ двухмерный массив вроде page на самом деле является массивом массивов. Иными словами, объявленный выше массив представляет собой одномерный массив из 30 элементов, базовым типом которого является одномерный массив символов из 100 элементов. Как правило, для программиста это не имеет значения, и он может рассматривать двухмерный массив просто как массив с двумя индексами. Однако иногда важно знать, чем этот массив является на самом деле. В качестве примера такого случая в следующем разделе мы рассмотрим функцию с параметром типа двухмерного массива.

Объявление многомерного массива

Синтаксис

тип имя_массива[размер_1][размер_2]...[размер_последний];

Примеры

```
char page[30][100];
int matrix[2][3];
double three_d_picture[10][20][30];
```

Объявление массива в приведенной выше форме определяет по одному элементу массива для каждой комбинации его индексов. Так, второе из приведенных объявлений определяет следующие шесть элементов массива:

```
matrix[0][0], matrix[0][1], matrix[0][2],
matrix[1][0], matrix[1][1], matrix[1][2]
```

Параметры типа многомерных массивов

Следующее объявление двухмерного массива на самом деле определяет одномерный массив из 30 элементов, базовым типом которого является одномерный массив из 100 элементов.

```
char page[30][100];
```

Для того чтобы понять, как C++ работает с параметрами типа многомерных массивов, нужно рассматривать многомерный массив как одномерный массив массивов.

В качестве примера рассмотрим следующую функцию, принимающую массив, такой как page, и выводящую на экран его содержимое:

```
void display_page(const char p[][100], int size_dimension_1)
{
    for (int index1 = 0;
            index1 < size_dimension_1; index1++)
    {// Вывод одной строки:
        for (int index2 = 0; index2 < 100; index2++)
            cout << p[index1][index2];
        cout << endl;
    }
}
```

Заметьте, что для параметра типа двухмерного массива функции не известен один из его размеров, поэтому он должен быть передан в дополнительном параметре типа int. (Как и для обычного массива, компилятор позволяет задать в квадратных скобках после имени параметра число. Но это просто комментарий, игнорируемый компилятором.) Второй же размер (и все остальные, если их более двух) явно задается после имени формального параметра в квадратных скобках, вот так:

```
const char p[][100]
```

Если помнить, что многомерный массив — это массив массивов, данное правило обретает смысл. Поскольку двухмерный параметр типа массива

```
const char p[][100]
```

представляет собой массив массивов, первая пара квадратных скобок обозначает индекс этого массива и трактуется как индекс в обычном одномерном массиве. А число во второй паре скобок – это часть описания базового типа, которым является массив символов из 100 элементов, следовательно он должен быть задан явно.

Параметры типа многомерных массивов

Когда в заголовке или объявлении функции указывается, что ее параметром является многомерный массив, один из его размеров (соответствующий первому индексу) не задается. Поэтому его обычно передают при помощи отдельного параметра типа int. Ниже приведен пример объявления функции с параметром p типа двухмерного массива.

```
void get_page(char p[][100], int size_dimension_1);
```

Пример: программа, использующая двухмерный массив

В листинге 10.15 приведена программа для работы с оценками студентов маленькой группы. Оценки хранятся в двухмерном массиве, названном grade. В группе четыре студента, каждый прошел по три теста. На рис. 10.3 показано, как организованы данные в массиве. Первый индекс используется для выбора студента, а второй – для выбора теста. Поскольку студенты и тесты нумеруются не с нуля, а с единицы, для получения индексов в массиве из их номеров всегда нужно вычитать единицу. Например, оценка студента номер 4 за тест номер 1 хранится в элементе grade[3][0] массива grade.

Листинг 10.15. Двухмерный массив

```
// Считывает оценки, полученные студентами за тесты,
// в массив grade (программный код, выполняющий ввод
// данных, в листинге не приведен).
// Вычисляет средний балл каждого студента и средний
// балл за каждый тест. Выводит оценки и средние значения.
#include <iostream>
#include <iomanip>
const int NUMBER_STUDENTS = 4, NUMBER_QUIZZES = 3;

void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[]);
// Предусловие: глобальные константы NUMBER_STUDENTS
// и NUMBER_QUIZZES определяют размеры массива grade.
// Каждый из элементов массива
// grade[st_num-1, quiz_num-1] содержит оценку
// студента номер st_num за тест номер quiz_num.
// Постусловие: каждый из элементов массива
// st_ave[st_num-1] содержит среднюю оценку
// студента номер stu_num.

void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[]);
// Предусловие: глобальные константы NUMBER_STUDENTS
// и NUMBER_QUIZZES определяют размеры массива grade.
```

продолжение

Листинг 10.15 (*продолжение*)

```
// Каждый из элементов массива
// grade[st_num-1, quiz_num-1] содержит оценку
// студента номер st_num за тест номер quiz_num.
// Постусловие: каждый из элементов массива
// quiz_ave[quiz_num-1] содержит среднюю оценку за
// тест номер quiz_num.

void display(const int grade[][NUMBER_QUIZZES],
             const double st_ave[], const double quiz_ave[]);
// Предусловие: глобальные константы NUMBER_STUDENTS
// и NUMBER_QUIZZES определяют размеры массива grade.
// Каждый из элементов массива
// grade[st_num-1, quiz_num-1] содержит оценку
// студента номер st_num за тест номер quiz_num.
// Каждый из элементов массива st_ave[st_num-1]
// содержит среднюю оценку студента номер stu_num.
// Каждый из элементов массива quiz_ave[quiz_num-1]
// содержит среднюю оценку за тест номер quiz_num.
// Постусловие: все данные из массивов grade, st_ave
// и quiz_ave выведены на экран.

int main()
{
    using namespace std;
    int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
    double st_ave[NUMBER_STUDENTS];
    double quiz_ave[NUMBER_QUIZZES];

... Здесь располагается код, заполняющий массив, но мы его не приводим. ...

    compute_st_ave(grade, st_ave);
    compute_quiz_ave(grade, quiz_ave);
    display(grade, st_ave, quiz_ave);
    return 0;
}

void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[])
{
    for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
    {// Обработка одного номера студента st_num:
        double sum = 0;
        for (int quiz_num = 1;
                 quiz_num <= NUMBER_QUIZZES; quiz_num++)
            sum = sum+grade[st_num-1][quiz_num-1];
        // sum содержит сумму оценок студента номер st_num.
        st_ave[st_num-1] = sum/NUMBER_QUIZZES;
        // Элементу массива st_ave[st_num-1] присвоена средняя
        // оценка студента номер st_num.
    }
}

void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[])
```

```
{
    for (int quiz_num = 1;
            quiz_num <= NUMBER_QUIZZES; quiz_num++)
    {// Обрабатываем один тест (для всех студентов):
        double sum = 0;
        for (int st_num = 1;
                st_num <= NUMBER_STUDENTS; st_num++)
            sum = sum+grade[st_num-1][quiz_num-1];
        // sum содержит сумму всех оценок за тест quiz_num.
        quiz_ave[quiz_num-1] = sum/NUMBER_STUDENTS;
        // Элементу массива quiz_ave[quiz_num-1] присвоена средняя
        // оценка за тест номер quiz_num.
    }
}

// Использует библиотеки iostream и iomanip.
void display(const int grade[][NUMBER_QUIZZES],
             const double st_ave[], const double quiz_ave[])
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);

    cout << setw(10) << "Student"
         << setw(5) << "Ave"
         << setw(12) << "Quizzes\n";
    for (int st_num = 1;
            st_num <= NUMBER_STUDENTS; st_num++)
    {// Вывод для одного st_num:
        cout << setw(10) << st_num
             << setw(5) << st_ave[st_num-1] << " ";
        for (int quiz_num = 1;
                quiz_num <= NUMBER_QUIZZES; quiz_num++)
            cout << setw(5)
                 << grade[st_num-1][quiz_num-1];
        cout << endl;
    }

    cout << "Quiz averages = ";
    for (int quiz_num = 1;
            quiz_num <= NUMBER_QUIZZES; quiz_num++)
        cout << setw(5) << quiz_ave[quiz_num-1];
    cout << endl;
}
```

Пример диалога (диалог заполнения массива не приведен)

```
Student    Ave       Quizzes
      1   10.0       10   10   10
      2    1.0        2    0    1
      3    7.7        8    6    9
      4    7.3        8    4   10
 Quiz averages =    7.0  5.0  7.5
```

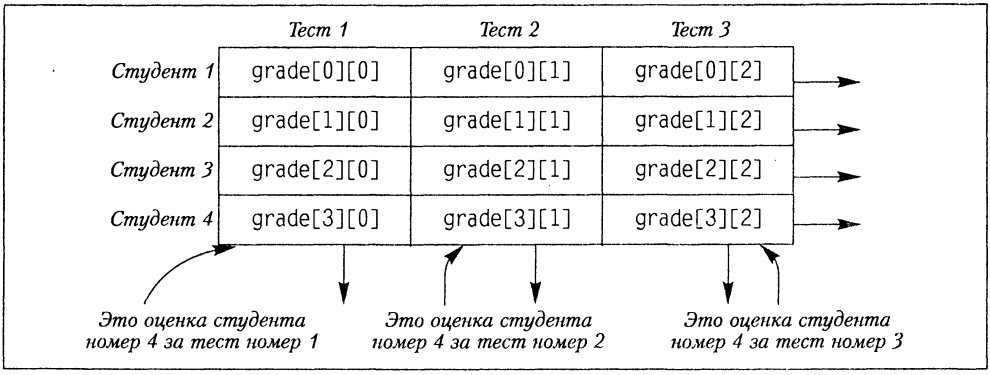


Рис. 10.3. Организации данных в массиве grade

Кроме двухмерного, в нашей программе используются два обыкновенных одномерных массива. В массив st_ave записываются средние оценки студентов. Например, элементу st_ave[0] программа присваивает среднюю оценку студента номер 1, элементу st_ave[1] — среднюю оценку студента номер 2 и т. д. В массив quiz_ave записываются средние оценки за тесты. Так, элементу quiz_ave[0] программа присваивает среднюю оценку за тест номер 1, элементу quiz_ave[1] — среднюю оценку за тест номер 2 и т. д. На рис. 10.4 показана взаимосвязь между массивами grade, st_ave и quiz_ave, а также приведен пример данных для массива grade. Эти данные определяют значения, которые программа помещает в массивы st_ave и quiz_ave.

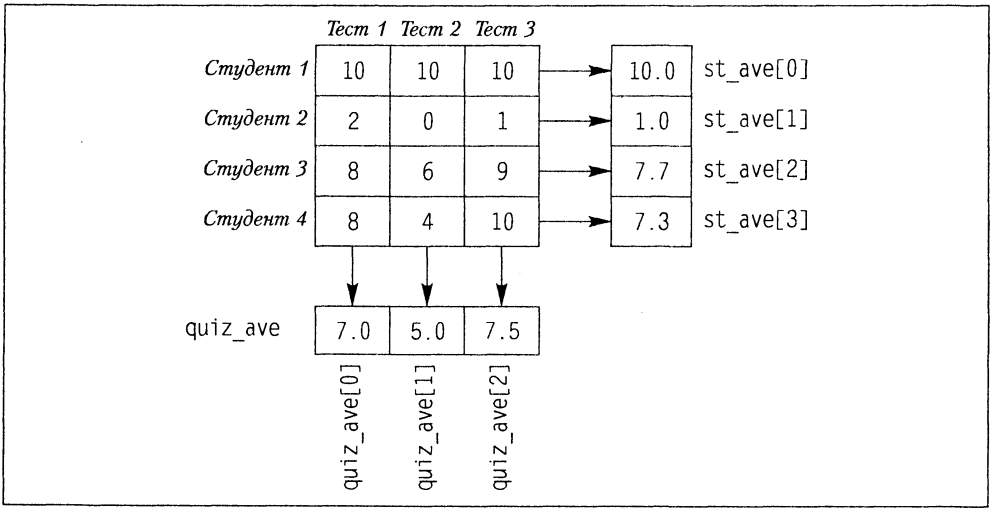


Рис. 10.4. Взаимосвязь между массивами grade, st_ave и quiz_ave

Полная программа для заполнения массива grade, вычисления и вывода средних оценок для каждого студента и для каждого теста приведена в листинге 10.15.

Размеры массива в ней объявлены как глобальные именованные константы. Поскольку процедуры этой программы не предназначены для повторного использования в каких-либо других программах, эти глобальные константы применяются прямо в теле процедур, а не передаются им в качестве параметров. Код заполнения массива настолько прост, что в листинге он опущен.

Ловушка: запятые между индексами массива

Обратите внимание, что в листинге 10.15 обращение к элементам двухмерного массива grade выполняется с использованием двух пар квадратных скобок: grade[st_num-1][quiz_num-1]. В некоторых языках программирования индексы указываются в одной паре квадратных скобок через запятую, вот так: grade[st_num-1, quiz_num-1], но в C++ это недопустимо. Если в программе на языке C++ применить такую запись, компилятор, скорее всего, не выведет сообщение об ошибке, но программа будет работать неправильно.

Упражнения для самопроверки

24. Что выведет следующий код:

```
int my_array[4][4], index1, index2;
for (index1 = 0; index1 < 4; index1++)
    for (index2 = 0; index2 < 4; index2++)
        my_array[index1][index2] = index2;
for (index1 = 0; index1 < 4; index1++)
{
    for (index2 = 0; index2 < 4; index2++)
        cout << my_array[index1][index2] << " ";
    cout << endl;
}
```

25. Напишите программный код для заполнения объявленного ниже массива a числами, введенными с клавиатуры. Числа вводятся по пять в строке в четырех строках (ваше решение не должно зависеть от того, каким образом вводимые числа будут разбиты на строки).

```
int a[4][5];
```

26. Напишите определение функции типа void с именем echo так, чтобы вызов

```
echo(a, 4);
```

выполнял эхо-вывод входных данных из упражнения 25 в формате, описанном в этом же упражнении (четыре строки по пять чисел).

Резюме

- Назначение массивов — хранение и обработка наборов однотипных данных.
- Элементы массива используются так же, как простые переменные того же типа, что и базовый тип массива.
- Цикл for удобен для перебора элементов массива и выполнения над ними однообразных операций.

- Типичной ошибкой при работе с массивами является попытка доступа к несуществующему элементу массива. Всегда проверяйте, правильные ли индексы используются на первой и последней итерации цикла обработки массива.
- Формальный параметр типа массива передается не по ссылке и не по значению, а особым способом, таким как передача по ссылке, когда в теле функции можно изменять элементы переданного ей массива.
- Элементы массива хранятся в памяти компьютера последовательно, занимая непрерывную область памяти. Когда массив используется в качестве аргумента функции, в нее передается только адрес его первого элемента (то есть элемента с индексом 0). Поэтому функции с параметрами типа массива обычно имеют еще один формальный параметр типа `int`, в котором задается размер массива.
- При использовании частично заполненного массива программе нужна дополнительная переменная типа `int` для отслеживания количества применяемых элементов.
- Для того чтобы указать компилятору, что аргумент типа массива не должен изменяться функцией, перед ее параметром нужно вставить квалификатор `const`. Такой параметр называется константным.
- Базовым типом массива может быть структура или класс, а массив может быть переменной-членом структуры или класса.
- Если в программе требуется массив, у которого будет более одного индекса, можно объявить многомерный массив, представляющий собой массив массивов.

Ответы к упражнениям для самопроверки

1. Оператор `int a[5];` является объявлением массива из пяти элементов. Выражение `a[4]` — это обращение к массиву, объявленному приведенным выше оператором. Оно используется для доступа к элементу с индексом 4, то есть к последнему, пятому элементу массива `a`.
2. а) `score`;

   б) `double`;

   в) `5`;

   г) от `0` до `4`;

   д) любой из элементов массива `score[0]`, `score[1]`, `score[2]`, `score[3]`, `score[4]`.
3. а) слишком много инициализационных значений;

   б) все правильно, размер массива равен 4;

   в) все правильно, размер массива равен 4.
4. `abc`.
5. `1.1 2.2 3.3`
   `1.1 3.3 3.3`

(Помните, что индексация начинается с нуля, а не с единицы.)

6. `0 2 4 6 8 10 12 14 16 18`
   `0 4 8 12 16`

7. Элементами массива `sample_array` являются `sample_array[0] ... sample_array[9]`, но этот код пытается заполнить элементы от `sample_array[1]` до `sample_array[10]`. Индекс 10 выходит за пределы допустимого диапазона.

8. Выход индекса за пределы допустимого диапазона. Когда переменная `index` равна 9, выражение `index+1` равно 10, поэтому выражение `a[index+1]` эквивалентно выражению `a[10]`, то есть обращению к элементу с недопустимым индексом. Для того чтобы исправить эту ошибку, нужно следующим образом изменить первую строку цикла `for`:

```
for (int index = 0; index < 9; index++)
```

9. ```
   int i, a[20];
   cout << "Enter 20 numbers:\n";
   for (i = 0; i < 20; i++)
       cin >> a[i];
   ```

10. Массив будет занимать в памяти 14 байт. Адрес его элемента `your_array[3]` равен 1006.

11. Допустимы следующие вызовы функции:

```
tripler(number);
tripler(a[2]);
tripler(a[number]);
```

Приведенные ниже вызовы недопустимы:

```
tripler(a[3]);
tripler(a);
```

В первом из них используется недопустимый индекс, а во втором он не задан вообще. Функции `tripler` нельзя передавать целый массив, как делается во втором вызове. Это ограничение касается конкретной функции, но в общем случае можно определить функцию так, чтобы она принимала в качестве аргумента целый массив. О том, как это сделать, рассказывается в разделе «Массивы в качестве аргументов функций».

12. В цикле обрабатываются элементы массива от `b[1]` до `b[5]`, но индекс 5 для этого массива недопустим. Допустимыми индексами являются 0, 1, 2, 3 и 4. Вот правильная версия кода этого упражнения:

```
int b[5] = {1, 2, 3, 4, 5};
for (int i = 0; i < 5; i++)
    tripler(b[i]);
```

13. ```
    void one_more(int a[], int size)
    // Предусловие: size – это объявленный размер массива a.
    // Элементы массива  от a[0] до a[size-1] содержат значения.
    // Постусловие: для всех элементов массива
    // a значение элемента a[index] увеличено на 1.
    {
        for (int index = 0; index < size; index++)
    ```

```
        a[index] = a[index]+1;
}
```

14. Допустимы следующие вызовы функции:

```
too2(my_array, 29);
too2(my_array, 10);
too2(your_array, 100);
```

Вызов

```
too2(my_array, 10);
```

допустим, но заполняет только первые десять элементов массива my_array. Если требуется именно это, он совершенно правилен.

Следующие вызовы функции недопустимы:

```
too2(my_array, 55);
"Hey too2. Please, come over here."
too2(my_array[3], 29);
```

Первый из них неправилен, поскольку в нем задано слишком большое значение второго аргумента. Во втором операторе отсутствует завершающая точка с запятой, но это еще не все: сам оператор представляет собой текстовую строку, заключенную в парные кавычки, поэтому слово too2 в нем не может рассматриваться как вызов функции. Третий вызов неправилен, потому что в нем в качестве аргумента используется элемент массива, а функции требуется целый массив.

15. Параметр типа массива функции output можно определить как константный, потому что функция не должна изменять значения его элементов. Параметр функции drop_odd нельзя сделать константным, так как эта функция может изменять значения некоторых элементов массива.

```
void output(const double a[], int size);
// Предусловие: элементы массива от a[0]
// до a[size-1] содержат значения.
// Постусловие: выведены все значения от a[0] до a[size-1].

void drop_odd(int a[], int size);
// Предусловие: элементы массива от a[0]
// до a[size-1] содержат значения.
// Постусловие: все нечетные значения от a[0] до a[size-1]
// заменены нулями.
```

16.
```
int out_of_order(double array[], int size)
{
    for(int i = 0; i < size-1; i++)
        if (array[i] > array[i+1]) // Перебирает a[i+1] для всех i.
            return i+1;
    return -1;
}
```

17.
```
#include <iostream>
using namespace std;
const int DECLARED_SIZE = 10;
int main()
{
```

```
    cout << "Enter up to ten nonnegative integers.\n"
         << "Place a negative number at the end.\n";
    int number_array[DECLARED_SIZE], next, index = 0;
    cin >> next;
    while ( (next >= 0) && (index < DECLARED_SIZE) )
    {
        number_array[index] = next;
        index++;
        cin >> next;
    }
    int number_used = index;
    cout << "Here they are back at you:";
    for (index = 0; index < number_used; index++)
        cout << number_array[index] << " ";
    cout << endl;
    return 0;
}
```

18.
```
#include <iostream>
using namespace std;
const int DECLARED_SIZE = 10;
int main()
{
    cout << "Enter up to ten letters"
         << " followed by a period:\n";
    char letter_box[DECLARED_SIZE], next;
    int index = 0;
    cin >> next;
    while ( (next != '.') && (index < DECLARED_SIZE) )
    {
        letter_box[index] = next;
        index++;
        cin >> next;
    }
    int number_used = index;
    cout << "Here they are backwards:\n";
    for (index = number_used-1; index >= 0; index--)
        cout << letter_box[index];
    cout << endl;
    return 0;
}
```

19.
```
bool search(const int a[], int number_used,
                           int target, int& where)
{
    int index = 0;
    bool found = false;
    while ((!found) && (index < number_used))
        if (target == a[index])
            found = true;
        else
            index++;
    // Если значение target найдено, то
    // found == true и a[index] == target.
    if (found)
        where = index;
    return found;
}
```

20.
```
struct Score
{
    int home_team;
    int opponent;
};
Score game[10];
```

21.
```
// Считывает 5 денежных сумм, удваивает каждую из них
// и выводит результаты на экран.
#include <iostream>
#include "money.h"
int main()
{
    using namespace std;
    Money amount[5];
    int i;
    cout << "Enter 5 amounts of money:\n";
    for (i = 0; i < 5; i++)
        cin >> amount[i];
    for (i = 0; i < 5; i++)
        amount[i] = amount[i]+amount[i];
    cout << "After doubling, the amounts are:\n";
    for (i = 0; i < 5; i++)
        cout << amount[i] << " ";
    cout << endl;
    return 0;
}
```

(Нельзя написать 2*amount[i], поскольку оператор * не перегружен для операндов типа Money.)

22. Это ответ для двух упражнений, данного и следующего. Определение класса нужно изменить таким образом

```
namespace tlistsavitch
{
    class TemperatureList
    {
    public:
       TemperatureList();

       int get_size() const;
       // Возвращает количество значений температуры в списке.

       void add_temperature(double temperature);

       double get_temperature(int position) const;
       // Предусловие: 0 <= position < get_size().
       // Возвращает значение температуры по заданному
       // индексу. Первое значение температуры имеет индекс 0.

       bool full() const;

       friend ostream& operator <<(ostream& outs,
                   const TemperatureList& the_object);
```

```
    private:
        double list[MAX_LIST_SIZE];// Значения температуры по  Фаренгейту.
        int size; // Количество заполненных элементов массива.
    };
} // Пространство имен tlistsavitch.
```

Для экономии места мы удалили некоторые комментарии, приведенные в листинге 10.13, но вы должны включить их в ответ.

Кроме того, нужно добавить следующие определения функций-членов:

```
int TemperatureList::get_size() const
{
    return size;
}

// Используем библиотеки классов iostream и cstdlib.
double TemperatureList::get_temperature (int position) const
{
    if ( (position >= size) || (position < 0) )
    {
        cout << "Error:"
             << " reading an empty list position.\n";
        exit(1);

    }
    else
    {
        return (list[position]);
    }
}
```

23. См. ответ 22.

24.
```
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

25.
```
int a[4][5];
int index1, index2;
for (index1 = 0; index1 < 4; index1++)
    for (index2 = 0; index2 < 5; index2++)
        cin >> a[index1][index2];
```

26.
```
void echo(const int a[][5], int size_of_a)
// Выводит значения из массива a в size_of_a lines строках
// по 5 чисел в строке.
{
    for (int index1 = 0; index1 < size_of_a; index1++)
    {
        for (int index2 = 0; index2 < 5; index2++)
            cout << a[index1][index2] << " ";
        cout << endl;
    }
}
```

Практические задания

Задания с 1 по 6 не требуют использования структур или классов (хотя задание 6 более изящно реализуется с использованием структур.) В заданиях 7–10 предполагается использование структур и классов. Задания 11–14 должны быть реализованы на основе многомерных массивов, и в них не следует пользоваться ни структурами, ни классами (но в некоторых случаях использование класса или структуры может сделать решение более элегантным).

1. Программа, создаваемая в этом задании должна иметь три версии.

   Версия 1 (полностью интерактивный ввод-вывод). Напишите программу, считывающую с клавиатуры информацию о среднемесячном количестве осадков в городе, полученную путем статистического анализа данных за много лет, и данных о количестве осадков, выпавших за каждый из 12 месяцев последнего года. Программа должна выводить аккуратно отформатированную сравнительную таблицу, показывающую количество осадков за каждый из последних 12 месяцев и разность между этим значением и среднемесячным количеством осадков за этот же период. Средние значения задаются за январь, февраль и т. д. по порядку. Для получения реального количества осадков за последние 12 месяцев программа сначала спрашивает, какой сейчас месяц, а затем просит ввести 12 значений за последние 12 месяцев. Выходные данные должны быть правильно распределены по месяцам.

   Существует множество способов работы с названиями месяцев. Проще всего пронумеровать месяцы, применяя числа от 1 до 12, и выполнять преобразование в текстовые названия только при выводе данных. Для этого удобно использовать оператор `switch`. Ввод месяца может выполняться по-разному – главное, чтобы было удобно пользователю.

   Завершив работу над описанной программой, создайте ее расширенную версию, выводящую сравнительную диаграмму среднего и фактического количества осадков за каждый из прошедших 12 месяцев. Она должна быть похожа на гистограмму, приведенную в листинге 10.7, с той разницей, что для каждого месяца будет содержать по два столбца, показывающих среднемесячное количество осадков и количество осадков за заданный месяц истекшего года. Программа должна запрашивать у пользователя, в каком виде он хочет получить результаты: в виде таблицы или в виде гистограммы, и выводить их в указанном формате. Включите в программу цикл, позволяющий пользователю произвольное количество раз выбирать то один, то другой формат выходных данных, до тех пор, пока он не захочет закончить работу.

   Версия 2 (сочетающая интерактивный и файловый вывод). Более сложная версия программы позволяет пользователю указать, что таблица или гистограмма должна быть выведена в файл. В остальном она идентична первой версии. Для того чтобы запрограммировать ввод имени файла, нужно изучить материал, приведенный в факультативном разделе «Использование имен файлов в качестве входных данных» главы 5.

Версия 3 (файловый ввод-вывод). Эта версия подобна первой с той разницей, что входные данные вводятся из файла, и вывод тоже направляется в файл. Поскольку в этом случае программа не может взаимодействовать с пользователем, в ней отсутствует цикл для повторения вывода, так что в файл выводятся и таблица, и диаграмма.

2. Напишите функцию `delete_repeats`, получающую в качестве параметра частично заполненный массив символов и удаляющую из него повторяющиеся символы. Для передачи этого массива требуется два аргумента, поэтому у функции имеется два формальных параметра: массив и параметр типа `int`, задающий количество заполненных элементов массива. При удалении символа все остальные сдвигаются к началу массива, заполняя пробел. В результате увеличивается число неиспользуемых элементов в конце массива. Второй параметр, где задается исходное количество заполненных элементов массива, передается по ссылке, чтобы функция могла в нем вернуть результирующее количество заполненных элементов после удаления повторяющихся символов.

   В качестве примера рассмотрим следующий код:

```
char a[10];
a[0] = 'a';
a[1] = 'b';
a[2] = 'a';
a[3] = 'c';
int size = 4;
delete_repeats(a, size);
```

   После его выполнения элемент `a[0]` массива `a` будет содержать значение `'a'`, элемент `a[1]` — значение `'b'`, элемент `a[2]` — значение `'c'`, а переменная `size` — значение 3. (Содержимое элемента `a[3]` нас больше не интересует, поскольку он относится к неиспользуемой части массива.) Предполагается, что частично заполненный массив содержит только буквы нижнего регистра. Включите функцию в подходящую отладочную программу.

3. Стандартное отклонение последовательности чисел — это критерий отличия данных чисел от их среднего значения. Если стандартное отклонение мало, числа отличаются от среднего значения на малую величину, если же оно велико, это означает, что их разброс велик. Стандартное отклонение S последовательности из N чисел x_i определяется следующей формулой:

$$S = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}}$$

   где $\bar{x}$ — среднее арифметическое N чисел x_1, x_2, ... Определите функцию, принимающую в качестве аргумента частично заполненный массив чисел и возвращающую их стандартное отклонение. У функции имеются два формальных параметра: массив и параметр типа `int`, в котором задается количество его заполненных элементов. Числа в массиве принадлежат к типу `double`. Включите функцию в подходящую отладочную программу.

4. Напишите программу, считывающую последовательность целых чисел в массив с базовым типом `int`. Она должна предоставлять возможность ввода данных

и с клавиатуры, и из файла в зависимости от желания пользователя. Если он выберет ввод из файла, программа должна запросить у него имя файла. Последовательность может содержать не более 49 чисел. Программа должна подсчитывать количество чисел в последовательности и выводить результаты в два столбца. В первом содержатся различающиеся элементы массива, а во втором — количество вхождений каждого элемента в массив. Результаты должны быть отсортированы по убыванию значений первого столбца.

Например, для входных данных:

```
-12 3 -12 4 1 1 -12 1 -1 1 2 3 4 2 3 -12
```

программа должна вывести:

```
 N  Count
 4  2
 3  3
 2  2
 1  4
-1  1
-12 4
```

5. В данной главе описан алгоритм сортировки методом выбора. Мы предлагаем еще один алгоритм, называемый сортировкой методом вставки. В определенном смысле он противоположен первому, поскольку последовательно выбирает элементы массива и помещает их в уже отсортированный подмассив, который находится в начале либо в конце сортируемого массива. Все производится таким образом, чтобы не нарушать отсортированности подмассива.

   Подлежащий сортировке массив делится на две части — отсортированный и неотсортированный подмассивы. Из второго подмассива по очереди выбираются элементы и помещаются в отсортированный подмассив так, чтобы не нарушать условие его сортировки. Напишите функцию и отладочную программу, реализующие сортировку методом вставки. Тщательно протестируйте программу.

   Пример и подсказка. В реализации алгоритма потребуется внешний цикл, выбирающий последовательные элементы неотсортированного подмассива, и вложенный цикл, помещающий выбранный элемент в правильную позицию в отсортированном подмассиве.

   Первоначально отсортированный подмассив пуст, а неотсортированный содержит все элементы массива:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	10

   Выбираем первый элемент, `a[0]` (то есть 8), и помещаем его в первую позицию. Внутреннему циклу в этом случае ничего делать не нужно. После этого массив и подмассив выглядят так:

отсортированный	неотсортированный								
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
8	6	10	2	16	4	18	14	12	10

Первым элементом неотсортированного подмассива является a[1] (имеющий значение 6). Его нужно вставить в отсортированный подмассив так, чтобы не нарушить условие сортировки. В данный момент это условие нарушается, и внутренний цикл должен поменять местами значения элементов a[0] и a[1]. Вот что получится в результате:

отсортированный		неотсортированный							
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10	2	16	4	18	14	12	10

Заметьте, что отсортированный подмассив увеличился на один элемент.

Данный процесс повторяется для первого элемента неотсортированного массива, то есть для элемента a[2], который нужно вставить в отсортированный подмассив так, чтобы он остался отсортированным. Поскольку элемент a[2] и так находится на месте (то есть он больше наибольшего элемента отсортированного подмассива и поэтому располагается последним), внутреннему циклу ничего делать не нужно. Вот результат этой итерации внешнего цикла:

отсортированный			неотсортированный						
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10	2	16	4	18	14	12	10

Снова выбираем первый элемент неотсортированного подмассива – a[3]. На этот раз внутренний цикл должен менять местами значения до тех пор, пока a[3] не окажется на нужном месте, вот так:

отсортированный				неотсортированный					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8	10<---->	2	16	4	18	14	12	10

отсортированный				неотсортированный					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6	8<---->	2	10	16	4	18	14	12	10

отсортированный				неотсортированный					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
6<-------->	2	8	10	16	4	18	14	12	10

отсортированный				неотсортированный					
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
2	6	8	10	16	4	18	14	12	10

Алгоритм продолжает свою работу до тех пор, пока неотсортированный массив не станет пустым, а в отсортированном не окажутся все элементы исходного массива.

6. Массив может использоваться для хранения больших целых чисел в виде последовательности цифр. Например, целое число 1234 можно записать в массив так: элементу a[0] присваивается значение 1, элементу a[1] — значение 2, элементу a[2] — значение 3 и элементу a[3] — значение 4. Однако для выполнения приведенной ниже задачи удобнее хранить цифры в обратном порядке: 4 — в a[0], 3 — в a[1], 2 — в a[2] и 1 — в a[3].

   Вам нужно написать программу, считывающую с клавиатуры два положительных целых числа длиной не более 20 знаков и выводящую их сумму. Эта программа считывает цифры как значения типа char, так что, например, число 1234 будет прочитано как '1', '2', '3' и '4'. После прочтения всего числа символы заменятся значениями типа int. Цифры записываются в массив, и после их ввода с клавиатуры имеет смысл заменить порядок элементов этого массива обратным. (Изменять ли порядок элементов — решать вам. Можно хранить числа любым из двух способов, и каждый из них имеет свои достоинства и недостатки.)

   Программа должна выполнять сложение тем же способом, каким вы складываете числа на бумаге. Полученный результат необходимо записать в массив размером в 20 элементов, после чего вывести на экран. Если результат сложения будет содержать более 20 цифр, программа должна вывести сообщение о «переполнении разрядной сетки». Программа должна позволять изменять максимальную длину обрабатываемых целых чисел путем изменения единственной глобальной константы. Включите в программу цикл, позволяющий пользователю вводить и складывать числа до тех пор, пока он не захочет завершить работу.

7. Напишите программу, считывающую строку текста и выводящую все содержащиеся в ней буквы и количество вхождений каждой буквы. Буквы должны выводиться в порядке уменьшения частоты их вхождения. Воспользуйтесь массивом с базовым типом struct, каждый элемент которого содержит букву и цифру. Предполагается, что входная строка вводится в нижнем регистре. Например, для строки

   ```
   do be do bo.
   ```

   должно быть выведено следующее:

   ```
   Letter  Number of Occurrences
       o   3
       d   2
       b   2
       e   1
   ```

   Программа должна будет отсортировать массив в порядке убывания целых чисел, представляющих собой количество вхождений в строку каждой буквы. Для этого можно воспользоваться модифицированной версией функции sort, приведенной в листинге 10.10. Для данной программы функцию придется изменить. Создайте два варианта: в первом ввод-вывод должен производиться с использованием клавиатуры и экрана, во втором — с применением двух файлов.

8. Напишите программу для оценки комбинации карт на руках у игрока в пятикарточный покер: ничего, пара (две одинаковых карты разной масти), две пары, тройка (три одинаковых карты разной масти), стрит (карты идущие по старшинству непосредственно одна за другой, не обязательно одной масти), флэш (все карты одной масти, например все пики), фул хауз (пара и тройка), карэ (четыре одинаковых карты разной масти), флэш стрит (флэш и стрит одновременно). Для хранения комбинации карт на руках у игрока используйте массив структур. Структура должна содержать две переменные: значение карты и масть. В программе должен присутствовать цикл, позволяющий пользователю оценивать новые комбинации до тех пор, пока он не захочет прекратить работу.

9. Напишите программу для вычисления баланса чековой книжки. С целью поиска всех чеков, не оплаченных со времени последнего вычисления баланса, она будет считывать следующую информацию : номер чека, сумму, оплачен ли чек. Используйте массив, базовым типом которого является класс. Этот класс будет служить для описания чека. Он должен содержать три переменные для трех указанных значений. Сумма чека должна быть представлена переменной-членом класса `Money` (определенного в листинге 10.11). Таким образом, один класс в программе будет использоваться как составная часть другого. Класс, который описывает чек, должен содержать аксессоры и мутаторы, а также конструкторы и функции для ввода и вывода информации о чеке.

   Кроме информации о чеках, программа считывает суммы всех депозитов, а также старый и новый баланс счета (указанный банком). Для хранения депозитов можно использовать еще один массив. Новый баланс должен быть равен старому плюс сумма всех депозитов минус сумма всех оплаченных чеков.

   Программа выводит общую сумму оплаченных чеков, общую сумму депозитов, вычисленный новый баланс и разность между этим значением и новым балансом, указанным банком. Кроме того, она выводит два списка чеков: чеки, оплаченные со времени последнего вычисления баланса чековой книжки, и неоплаченные чеки. Оба списка выводятся отсортированными по возрастанию номеров.

10. Определите класс `List` для хранения списка значений типа `double`. Используйте в качестве образца класс `TemperatureList`, приведенный в листингах 10.13 и 10.14, но в вашем классе ничто не должно говорить о том, каков смысл хранящихся в нем значений. Они могут представлять любые данные, для которых подходит тип `double`. Включите в свой класс дополнительные элементы, описанные в упражнениях 22 и 23. Измените имена функций-членов так, чтобы они не ассоциировались со значениями температуры.

    Добавьте в класс функцию-член `get_last`, не имеющую параметров и возвращающую последний элемент списка. Она не изменяет список и не должна вызываться, если тот пуст. Добавьте еще одну функцию-член, `delete_last`, удаляющую последний элемент списка. Это будет функция типа `void`. Обратите внимание, что при удалении последнего элемента списка должно соответственно изменяться значение переменной-члена `size`. Если функция `delete_last`

вызвана для пустого списка, она ничего не делает. Определение класса распределите между файлами интерфейса и реализации, как это сделано для класса `TemperatureList` в листингах 10.15 и 10.16. Напишите программу, чтобы произвести тщательное тестирование класса `List`.

11. Напишите программу для игры в крестики-нолики, поочередно запрашивающую ходы у игрока А и у игрока Б. Решетку для игры программа должна выводить так:

```
1 2 3
4 5 6
7 8 9
```

Каждый игрок вводит цифру, соответствующую клетке, в которую он хочет поставить свой значок. Вот пример вывода программы после двух ходов каждого из игроков:

```
X X O
4 5 6
O 8 9
```

12. Напишите программу для бронирования пассажирами мест в самолете. Это маленький самолет, и места в нем нумеруются так:

```
1  A B  C D
2  A B  C D
3  A B  C D
4  A B  C D
5  A B  C D
6  A B  C D
7  A B  C D
```

Программа должна выводить схему салона, в которой буквой X помечены те места, которые уже забронированы. Например, после продажи билетов на места 1A, 2B и 4C она должна вывести:

```
1  X B  C D
2  A X  C D
3  A B  C D
4  A B  X D
5  A B  C D
6  A B  C D
7  A B  C D
```

После вывода информации об имеющихся местах программа спрашивает пользователя о том, какое место он хочет забронировать. Тот вводит номер места, и программа выводит обновленную схему. Процесс повторяется до тех пор, пока все места не будут забронированы или пользователь не укажет, что хочет закончить работу. Если он введет номер уже забронированного места, программа должна сообщить об этом и предложить выбрать другое.

13. Напишите программу, у которой входные данные будут аналогичны входным данным программы из листинга 10.7. Она должна выводить такую же гистограмму, как программа из листинга, но не горизонтальную, а вертикальную. Здесь удобно воспользоваться двухмерным массивом.

14. Математик Джон Хортон Конвей изобрел «Игру жизни». Хотя она и не является игрой в традиционном смысле этого слова, но представляет собой интересный алгоритм с очень простыми правилами. Вам предлагается написать программу, реализующую этот алгоритм. Сначала программа должна запросить начальную конфигурацию описанной ниже популяции живых организмов. После этого программа следует определенным правилам (перечисленным ниже) и показывает изменение этой популяции.

    Основная единица популяции — модель организма, живущего в дискретном двухмерном мире. Теоретически этот мир бесконечен, но мы не можем работать с бесконечной моделью, поэтому ограничим размерность массива, представляющего этот мир, 80 столбцами и 22 строками. Если ваш экран позволяет отобразить больше символов, используйте большую размерность.

    Мир игры представляет собой матрицу, каждая ячейка которой может содержать один живой организм. Время измеряется поколениями. При смене поколений одни клетки рождаются, другие умирают. Это происходит согласно следующим правилам.

    - Каждая ячейка имеет восемь соседних ячеек. Соседними считаются ячейки, примыкающие к ней по горизонтали, по вертикали и по обеим диагоналям.
    - Если организм имеет не более одного соседа, он умирает от одиночества, если же он имеет более трех соседей, то умирает от перенаселенности.
    - Если к пустой ячейке примыкают ровно три занятых ячейки, в ней рождается новый организм.
    - Все рождения и смерти происходят одновременно. Смерть одного организма может вызвать рождение другого, но новый организм не может в то же мгновение занять место старого, а смерть одного организма не может предотвратить смерть другого (например, за счет сокращения локальной популяции).

    Примечания. Некоторые популяции разрастаются, другие перемещаются по плоскости. Рекомендуется использовать двухмерный массив с базовым типом `char` размерностью 80 столбцов на 22 строки для хранения структуры популяций последовательных поколений. Звездочкой отмечена ячейка с живым организмом, а пробелом — пустая ячейка. Если ваш экран вмещает больше строк и столбцов, используйте весь экран.

    Примеры:

    ```
    ***
    ```

    превращаются в

    ```
    *
    *
    *
    ```

    затем снова в

    ```
    ***
    ```

    и т. д.

Предложения. Найдите стабильные конфигурации, то есть сообщества, видоизменение которых циклически повторяется. Количество конфигураций в цикле называется периодом. Существуют и фиксированные конфигурации, которые не изменяются вообще. Одно из возможных заданий может заключаться в поиске таких конфигураций.

Подсказки. Определите функцию `generation` типа `void`, принимающую содержащий начальную конфигурацию массив `world` из 80 столбцов и 22 строк с элементами типа `char`. Функция сканирует этот массив и модифицирует ячейки, отмечая их согласно описанным выше правилам как живые или мертвые. Ячейки анализируются по очереди, и для каждой принимается решение: умерла ли она, продолжает ли жить или в ней родилась новая жизнь. В программе должна присутствовать функция `display`, принимающая массив `world` и выводящая его на экран. Между вызовами `generation` и `display` должна происходить небольшая задержка. Пусть программа формирует и выводит следующее поколение после нажатия клавиши Enter. При желании этот процесс можно автоматизировать.

Глава 11 Строки и векторы

Полоний: Что вы читаете, мой господин?
Гамлет: Слова, слова, слова.

Уильям Шекспир

В этой главе рассматриваются две темы, связанные с массивами: строки и векторы. Хотя они тесно соприкасаются, их можно изучать независимо друг от друга и в любом порядке.

В разделах 11.1 и 11.2 описаны два типа данных, значения которых представляют строки символов. Первый из них — массив базового типа `char`, в котором хранится последовательность символов строки, а конец строки отмечается нулевым символом `'\0'`. Это старый способ представления строк, унаследованный языком C++ от языка C. Строки данного типа, называемые *строками C*, все еще широко используются, и вам часто придется с ними сталкиваться. Например, строковые константы в кавычках, такие как `"Hello"`, реализуются в C++ как строки C.

Стандарт ANSI/ISO языка C++ включает более современный способ представления строк в виде объектов класса `string`. Это второй строковый тип, рассматриваемый в данной главе.

Векторы представляют собой массивы особого вида, размер которых может быть изменен непосредственно во время работы программы. Если длина обычного массива, созданного в программе C++, неизменна в течение всего времени работы программы, то длина вектора может меняться.

11.1. Массивы для хранения строк

Во всем нужно учитывать конец.

Жан де Лафонтен

В этом разделе описывается способ представления строк символов, унаследованный языком C++ от языка C. Хотя строки C несколько «старомодны», они попрежнему широко используются и являются неотъемлемой частью языка C++.

Строковые значения и строковые переменные C

Одним из способов представления строк является использование массива базового типа char. Например, строку "Hello" удобно представить как массив из шести индексированных переменных: пяти букв слова "Hello" и одного нулевого символа '\0', служащего маркером конца строки. Символ '\0' называется *нуль-символом* или *нулевым символом*, а когда он используется в качестве маркера конца строки — *нуль-терминатором*. При использовании таких маркеров программа может считывать массивы посимвольно и знать, когда следует остановиться. Строка, хранящаяся в описанном формате, называется строкой C.

В программе C++ нуль-символ записывается как '\0', то есть в виде двух символов, но на самом деле, подобно символу новой строки '\n', он является одним символом. Как и любое другое символьное значение, он может храниться в переменной типа char или элементе массива с базовым типом char.

> **Нуль-символ '\0'**
>
> Нуль-символ '\0' отмечает конец строки C, хранящейся в символьном массиве. Такой массив часто называют строковой переменной C. Хотя нуль-символ записывается в виде двух символов, это один символ, который может храниться в переменной типа char или элементе массива базового типа char.

Вам уже приходилось пользоваться строками C. Например, литеральная строка, подобная "Hello", хранится в виде строки C, хотя это редко имеет значение для программы.

Строковая переменная C представляет собой просто массив символов. Так, следующее объявление массива:

```
char s[10];
```

создает строковую переменную C, в которой может храниться строковое значение C, состоящее из десяти или менее символов.

Массив длиной десять символов вмещает строку из девяти символов и нуль-символ '\0', отмечающий ее конец.

Строковая переменная C является частично заполненным массивом символов. Подобно другим частично заполненным массивам, она содержит данные в идущих подряд элементах, начиная с нулевого. И в ней занято столько позиций, сколько нужно для хранения данных. Однако для отслеживания количества заполненных элементов массива ей не требуется отдельная переменная типа int. Информация о месте окончания строки содержится в ней самой — после последнего символа строки в строковой переменной C располагается специальный символ '\0'. Поэтому, если в переменной s содержится строка "Hi Mom!", элементы массива заполнены следующим образом:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
H	I		M	o	m	!	\0	?	?

Символ '\0' используется в качестве сигнального значения, отмечающего конец строки С. Если считывать символы строки, начиная с элемента s[0], затем s[1], s[2] и т. д., то дойдя до символа '\0', вы будете знать, что достигли конца строки. Поскольку этот символ всегда занимает один элемент массива, максимальная длина строки, которую может вместить массив, на единицу меньше объявленного размера этого массива.

Еще раз подчеркнем: в конце строковой переменной С обязательно должен располагаться нуль-символ '\0'. Отличие этого типа массива от остальных заключается не в структуре, а состоит в том, что он, представляя собой обычный массив символов, используется особым образом.

Как видно из следующего примера, строковую переменную С можно инициализировать при объявлении:

```
char my_message[20] = "Hi there.";
```

Обратите внимание на то, что строка С, присваиваемая строковой переменной С, не обязательно заполняет весь массив.

Объявление строковой переменной С

Строковая переменная С — это обычный массив символов, используемый особым образом. Она объявляется так же, как любой другой массив символов.

Синтаксис

```
char имя_массива[максимальный_размер_строки_С + 1];
```

Пример

```
char my_c_string[11];
```

Единица прибавляется для того, чтобы массив вмещал нуль-символ '\0', отмечающий конец хранящейся в массиве строки. В частности, строковая переменная my_c_string в приведенном выше примере вмещает строку С длиной в десять или менее символов.

Инициализируя строковую переменную С, можно опустить размер массива. C++ автоматически присвоит ей размер, который будет на единицу больше, чем длина заключенной в кавычки строки (один дополнительный элемент массива для символа '\0'). Например, объявление

```
char short_string[] = "abc";
```

эквивалентно объявлению

```
char short_string[4] = "abc";
```

Однако не путайте следующие две инициализации:

```
char short_string[] = "abc";
```

и

```
char short_string[] = {'a', 'b', 'c'};
```

Они не равнозначны. Первая помещает после символа 'c' символ '\0', а вторая этого не делает (она вообще не помещает в массив символ '\0' — ни после 'c', ни в каком-либо другом месте).

Поскольку строковая переменная C является массивом, она состоит из набора элементов, с которыми можно работать по отдельности. Предположим, что программа содержит такое объявление строковой переменной C:

```
char our_string[5] = "Hi";
```

Это объявление и инициализация массива символов, включающего следующие элементы: our_string[0], our_string[1], our_string[2], our_string[3] и our_string[4]. Для примера рассмотрим следующий фрагмент программы:

```
int index = 0;
while (our_string[index] != '\0')
{
    our_string[index] = 'X';
    index++;
}
```

Данный код изменяет строковое значение, хранящееся в переменной our_string, помещая в нее строку C, состоящую только из символов 'X'.

Инициализация строковой переменной C

Строковую переменную C можно инициализировать при объявлении, как в следующем примере:

```
char my_string[] = "Do Be Do";
```

Эта инициализация автоматически помещает в конец строки C символ '\0'.

Если в квадратных скобках не задано число, создается массив, длина которого на единицу больше длины помещаемой в него строки. Так, приведенный оператор объявляет массив my_string из девяти индексированных переменных (восемь для символов строки "Do Be Do" и один для нуль-символа '\0').

При работе с такими индексированными переменными нужно внимательно следить, чтобы символ '\0' не был заменен каким-нибудь другим значением, поскольку при отсутствии этого символа массив перестанет вести себя, как строковая переменная C. Например, код

```
char happy_string[7] = "DoBeDo";
happy_string[6] = 'Z';
```

изменяет массив happy_string так, что в нем больше не содержится строка C.

После выполнения приведенного кода массив happy_string будет по-прежнему содержать шесть букв строки "DoBeDo", но в нем не будет нуль-символа, отмечающего конец строки C (он заменен символом 'Z'). Многие функции для работы со строками требуют обязательного наличия нуль-символа '\0' и без него работают неправильно. В качестве еще одного примера можно рассмотреть приведенный выше цикл while, изменяющий символы строковой переменной our_string. Цикл заменяет символы до тех пор, пока не встретит символ '\0', а в случае его отсутствия он может «обработать» большой фрагмент памяти с непредсказуемыми последствиями. Цикл while можно переписать следующим образом:

```
int index = 0;
while ( (our_string[index] != '\0') && (index < SIZE) )
```

```
{
    our_string[index] = 'X';
    index++;
}
```

чтобы он не полагался на наличие символа '\0' и ни в коем случае не выходил за переделы массива.

В этом примере SIZE — именованная константа, равная объявленному размеру массива our_string.

Ловушка: использование операторов = и == со строками C

Строковые значения и переменные C отличаются от значений и переменных других типов данных, и многие операции языка C++ к ним неприменимы. Так, нельзя использовать строковую переменную C в операторе присваивания. Если же попытаться сравнить две строки C посредством оператора ==, результат будет не таким, как ожидалось. Это объясняется тем, что строки C являются массивами.

Присваивание значения строковой переменной C выполняется не так просто, как другим переменным C++. Скажем, следующий оператор присваивания:

```
char a_string[10];
a_string = "Hello";
```

недопустим.

Хотя в объявлении такой переменой можно пользоваться знаком равенства для ее инициализации, больше нигде в программе это не разрешается. Знак равенства в объявлении переменной

```
char happy_string[7] = "DoBeDo";
```

означает именно инициализацию, а не присваивание, которое выполняется иначе.

Существуют разные способы присваивания значения строковой переменной C, простейший из них заключается в вызове стандартной функции strcpy:

```
strcpy(a_string, "Hello");
```

Этот вызов присваивает переменной a_string строку "Hello". К сожалению, данная версия функции strcpy не проверяет, превышает ли размер строки размер строковой переменной.

Во многих реализациях C++ имеется более безопасная версия данной функции, называемая strncpy (с буквой n). У нее есть третий аргумент, в котором задается максимальное количество копируемых символов. Например:

```
char another_string[10];7
strncpy(another_string, a_string_variable, 9);
```

Вызов копирует максимум девять символов из строковой переменной a_string_variable (независимо от ее длины) в строковую переменную another_string.

Проверку эквивалентности двух строковых переменных C нельзя выполнять обычным способом (то есть с помощью оператора ==). Данный оператор можно

использовать со строками C для других целей. Поэтому если применить его для сравнения двух строк, результат окажется неверным, и при этом даже не будет выведено сообщение об ошибке. Сравнение двух строк C на эквивалентность выполняется с помощью стандартной функции strcmp. Вот так:

```
if (strcmp(c_string1, c_string2))
    cout << "The strings are NOT the same.";
else
    cout << "The strings are the same.";
```

Обратите внимание на то, что функция strcmp работает несколько необычно — она возвращает значение true, когда строки не равны. Эта функция сравнивает две строки посимвольно, и если для очередной пары символов код символа из строки c_string1 оказывается меньше кода символа из строки c_string2, она прекращает проверку и возвращает отрицательное число. В случае же когда код символа из строки c_string1 оказывается больше кода символа из строки c_string2, она возвращает положительное число. Ну а если строки одинаковы, функция strcmp возвращает 0. При сравнении символов двух строк используется *лексикографический* (*словарный*) порядок. Когда обе строки содержат символы одного регистра, этот порядок совпадает с алфавитным.

Таким образом, функция strcmp возвращает отрицательное число, положительное число или 0 в зависимости от того, окажется первая из переданных ей строк меньше, больше или равной второй с точки зрения их лексикографического порядка. Если использовать возвращаемый ею результат в качестве логического выражения в операторе if или в цикле для проверки равенства двух строк C, ненулевое значение будет преобразовано в true, а 0 — в false. При тестировании программ, выполняющих такую проверку, не забывайте об этой «логике наоборот».

Компиляторы C++, соответствующие стандарту ANSI/ISO, поддерживают более безопасную версию функции strcmp с третьим аргументом, в котором задается максимальное количество сравниваемых символов.

Функции strcpy и strcmp располагаются в библиотеке с заголовочным файлом <cstring>. Для их использования нужно добавить в начало программы следующую директиву:

```
#include <cstring>
```

Обе эти функции не требуют приведенной ниже (или подобной ей) директивы:

```
using namespace std;
```

но она может понадобиться в других частях программы.

Библиотека cstring

Для объявления и инициализации строк C не требуется никакой директивы include или using. Однако при обработке строк вы наверняка будете применять те или иные предопределенные функции из библиотеки cstring. Поэтому, решив воспользоваться строками C, лучше сразу включите в начало файла программы директиву

```
#include <cstring>
```

Функции из библиотеки cstring

Описания некоторых наиболее популярных функций из библиотеки с заголовочным файлом <cstring> приведены в табл. 11.1. Для их использования в начало файла программы нужно включить директиву

```
#include <cstring>
```

Мы уже рассказывали о функциях strcpy и strcmp, которые, как и другие функции из библиотеки cstring, не требуют директивы, подобной следующей:

```
using namespace std;
```

Однако в других частях программы она может быть нужна.

Еще одной удобной и полезной функцией является strlen, возвращающая длину заданной строки. Например, strlen("dobedo") возвращает 6, поскольку в строке "dobedo" содержится шесть символов.

Функция strcat выполняет конкатенацию двух строк С; иными словами, она создает более длинную строку путем слияния двух исходных строк, расположенных одна за другой. Первым ее аргументом должна быть строковая переменная С. Вторым может быть любое выражение, возвращающее строку С, скажем, строка в двойных кавычках. В качестве примера рассмотрим следующий код:

```
char string_var[20] = "The rain";
strcat(string_var, "in Spain");
```

Он изменяет значение переменной string_var, присваивая ей строку "The rainin Spain". Как видите, при слиянии строк нужно внимательно следить за расстановкой пробелов (между словами rain и in, скорее всего, должен быть пробел).

Просмотрев табл. 11.1, вы узнаете, что во многих, хотя и не во всех, компиляторах С++ имеются более защищенные версии функций strcpy, strcat и strcmp. Они имеют три аргумента и содержат дополнительную букву n (strncpy, strncat и strncmp).

Таблица 11.1. Некоторые стандартные функции для работы со строками С из библиотеки cstring

Функция	Описание	Предупреждение
strcpy(*строковая_переменная*, *строка*)	Копирует строковое значение С, заданное в аргументе *строка*, в строковую переменную С *строковая_переменная*	Не проверяет, поместится ли значение *строка* в переменную *строковая_переменная*
Strncpy(*строковая_переменная*, *строка*, *лимит*)	Похожа на strcpy, но копирует максимум *лимит* символов	Если значение *лимит* выбрано правильно, эта функция надежнее двухаргументной функции strcpy. Реализована не во всех версиях С++

продолжение

Таблица 11.1 *(продолжение)*

Функция	Описание	Предупреждение
strcat(*строковая_переменная*, *строка*)	Выполняет конкатенацию строк C, добавляя значение *строка* в конец строки, хранящейся в переменной *строковая_переменная*	Не проверяет, поместится ли объединенная строка в переменную *строковая_переменная*
strncat(*строковая_переменная*, *строка*, *лимит*)	Похожа на strcat, но добавляет максимум *лимит* символов	Если значение *лимит* выбрано верно, эта функция надежнее двухаргументной функции strcat. Реализована не во всех версиях C++
strlen(*строка*)	Возвращает целое число, равное длине строки *строка*. (Нуль-символ '\0' не учитывается.)	
strcmp(*строка_1*, *строка_2*)	Возвращает 0, если *строка_1* и *строка_2* одинаковы. Возвращает отрицательное значение, если *строка_1* меньше, чем *строка_2*; положительное значение, если *строка_1* больше, чем *строка_2*. При сравнении используется лексикографический порядок строк	Если *строка_1* и *строка_2* одинаковы, данная функция возвращает 0, который преобразуется в false. Обратите внимание, что это значение обратно тому, которое естественно было бы ожидать от функции
strncmp(*строка_1*, *строка_2*, *лимит*)	Похожа на strcmp, но добавляет максимум *лимит* символов	Если значение *лимит* выбрано правильно, эта функция надежнее двухаргументной функции strcmp. Реализована не во всех версиях C++

Строки C как аргументы и параметры

Строковая переменная C — это массив, поэтому когда она используется как параметр функции, тип данного параметра является типом массива. Функция может изменять значение строковой переменной так же, как значение любого другого параметра типа массива, поэтому вместе с ней обычно передается дополнительный аргумент типа int, в котором задается объявленный размер этой строковой переменной.

Если же функция использует значение строкового аргумента C, не изменяя его, дополнительный аргумент (в котором задавался бы объявленный размер массива или количество его заполненных элементов) не нужен. Конец строкового значения, хранящегося в строковой переменной C, отмечает нуль-символ '\0'.

Упражнения для самопроверки

1. Какие из следующих объявлений с инициализацией:

```
char string_var[10] = "Hello";
char string_var[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
char string_var[10] = {'H', 'e', 'l', 'l', 'o'};
char string_var[6] = "Hello";
char string_var[] = "Hello";
```

   являются эквивалентными?

2. Какая строка С будет находиться в переменной singing_string после выполнения следующего кода:

```
char singing_string[20] = "DoBeDo";
strcat(singing_string, " to you");
```

   Предполагается, что этот код выполняется в составе правильно написанной программы, содержащей директиву #include для библиотеки cstring.

3. Что неправильно (если что-либо неправильно) в следующем фрагменте кода:

```
char string_var[] = "Hello";
strcat(string_var, " and Good-bye.");
cout << string_var
```

   Предполагается, что этот код выполняется в составе правильно написанной программы, содержащей директиву #include для библиотеки cstring.

4. Допустим, что функция strlen, возвращающая длину строки, заданной в качестве аргумента, еще не определена. Попробуйте написать ее определение. У нее должен быть единственный аргумент — строка С, не добавляйте других аргументов, поскольку они не нужны.

5. Какова максимальная длина строки, помещающейся в строковую переменную, объявленную следующим образом:

```
char s[6];
```

   Поясните свой ответ.

6. Сколько символов включает каждая из следующих символьных и строковых констант:

   а) '\n';

   б) 'n';

   в) "Mary";

   г) "M";

   д) "Mary\n".

7. Символьные строки являются просто массивами базового типа char, почему же в этой главе говорилось, что не следует путать следующие объявления:

```
char short_string[] = "abc";
```

   и

```
char short_string[] = { 'a', 'b', 'c'};
```

8. В программе имеется такое объявление с инициализацией строковой переменной:

```
char our_string[15] = "Hi there!";
```

Напишите цикл, заполняющий эту переменную символами 'X', при этом длина строки не должна изменяться.

9. В программе имеется объявление строковой переменной (SIZE — именованная константа):

```
char our_string[SIZE];
```

Строковая переменная our_string получает значение далее в программе. Следующий цикл:

```
int index = 0;
while (our_string[index] != '\0')
{
    our_string[index] = 'X';
    index++;
}
```

заполняет хранящуюся в ней строку символами 'X', сохраняя длину строки неизменной. Данный цикл входит в состав правильно написанной программы. Почему этот код может разрушить содержимое памяти за границей массива? Модифицируйте цикл, чтобы защитить память от случайного изменения.

10. Напишите программный код, который копирует строковую константу "Hello" в объявленную ниже строковую переменную с использованием библиотечной функции.

```
char a_string[10];
```

Не забудьте включить в этот код заголовочный файл библиотеки, где находится используемая вами функция.

11. Какую строку выводит на экран следующий код:

```
char song[10] = "I did it ";
char franks_song[20];
strcpy ( franks_song, song );
strcat ( franks_song, "my way!");
cout << franks_song << endl;
```

Как всегда, предполагается, что он выполняется в составе правильно написанной программы.

12. Какие ошибки содержатся (если они есть) в следующем фрагменте кода:

```
char a_string[20] = "How are you? ";
strcat(a_string, "Good, I hope.");
```

Ввод и вывод строк С

Строки С можно выводить с помощью оператора вывода <<. Фактически мы уже делали это для литеральных строк, заключенных в двойные кавычки. Точно так же можно использовать и строковые переменные. Например:

```
cout << news << " Wow.\n";
```

Здесь news — строковая переменная С.

Строковые переменные C можно заполнять, используя оператор ввода >>, но нужно иметь в виду одно обстоятельство: как и для других типов данных, при вводе строк C пропускаются символы пробела, табуляции и перевода строки. Более того, на очередном из перечисленных символов процесс чтения входных данных останавливается. Рассмотрим следующий код:

```
char a[80], b[80];
cout << "Enter some input:\n";
cin >> a >> b;
cout << a << b << "END OF OUTPUT\n";
```

Если выполнить его в составе программы C++, получится примерно такой диалог:

```
Enter some input:
Do be do to you!
DobeEND OF OUTPUT
```

В строковые переменные a и b введенный текст добавляется по словам: в a помещается строка "Do", поскольку за ней следует пробел, а в b помещается строка "be", за которой тоже введен пробел.

Если вы хотите, чтобы программа прочитала всю строку с символами пробела и табуляции, то можно прочитать ее по частям с помощью оператора ввода >>, а потом собрать в единое целое. Но это, во-первых, утомительно, а во-вторых, вы не сможете различить символы пробела и табуляции и не будете знать их количества. Гораздо проще и надежнее воспользоваться стандартной функцией-членом getline, включенной в любой входной потоковый объект: и в cin, и во входной файловый поток. У функции getline есть два аргумента. Первым является строковая переменная, в которую вводятся данные, а вторым — целое число, обычно задающее объявленный размер строковой переменной. Функция интерпретирует второй аргумент как максимальное количество элементов, заданных в первом аргументе массива, которые можно заполнить символами. В качестве примера рассмотрим следующий код:

```
char a[80];
cout << "Enter some input:\n";
cin.getline(a, 80);
cout << a << "END OF OUTPUT\n";
```

При выполнении его в составе полной программы можно получить такой диалог:

```
Enter some input:
Do be do to you!
Do be do to you!END OF OUTPUT
```

Функция cin.getline считывает всю строку. Чтение закончится, когда будет достигнут конец строки, даже если она короче принимающей переменной. Рассмотрим такой программный код:

```
char short_string[5];
cout << "Enter some input:\n";
cin.getline(short_string, 5);
cout << short_string << "END OF OUTPUT\n";
```

При выполнении его в составе полной программы можно получить приведенный ниже диалог.

```
Enter some input:
dobedowap
dobeEND OF OUTPUT
```

Обратите внимание, что в строковую переменную short_string прочитаны четыре, а не пять символов, хотя вторым аргументом функции передано значение 5. Дело в том, что один элемент массива занят символом '\0'. Вы уже знаете, что им завершается каждая строка С и он всегда занимает один элемент массива.

Технологии ввода и вывода строк С посредством потоковых объектов cin и cout применимы и к файловым потоковым объектам. Входной поток cin можно заменить входным потоковым объектом, подключенным к файлу, а выходной поток cout можно заменить выходным потоковым объектом, подключенным к файлу. (О файловом вводе-выводе рассказывалось в главе 5.)

Функция getline

Функция-член потоковых объектов getline может использоваться для чтения входной строки и помещения ее в строковую переменную С.

Синтаксис

входной_поток.getline(*строковая_переменная*, *максимальное_количество_символов* + 1);

Из потока *входной_поток* считывается одна строка символов и помещается в переменную *строковая_переменная*. Если размер строки превышает *максимальное_количество_символов*, будет прочитана только ее начальная часть, равная по длине этому значению. (Единица прибавляется, так как каждая строка С содержит нуль-символ '\0', отмечающий ее конец, и поэтому строка в переменной *строковая_переменная* на единицу длиннее прочитанной строки.)

Пример

```
char one_line[80];
cin.getline(one_line, 80);
```

(Точно так же можно вводить данные из входного потока, подключенного к файлу.)

Упражнения для самопроверки

13. Рассмотрим следующий код (входящий в состав правильно написанной программы):

```
char a[80], b[80];
cout << "Enter some input:\n";
cin >> a >> b;
cout << a << '-' << b << "END OF OUTPUT\n";
```

Если диалог начинается таким образом:

```
Enter some input:
The
   time is now.
```

какой будет следующая выведенная программой строка?

14. Рассмотрим следующий код (входящий в состав правильно написанной программы):

```
char my_string[80];
cout << "Enter a line of input:\n";
cin.getline(my_string, 6);
cout << my_string << "<END OF OUTPUT";
```

Если диалог начинается таким образом:

```
Enter a line of input:
May the hair on your toes grow long and curly.
```

какой будет следующая выведенная программой строка?

Преобразование строк C в числа

Строка C "1234" и число 1234 — это не одно и то же. Первое выражение представляет собой последовательность символов, а второе — число. В жизни мы записываем их одинаково и не думаем о различии, но в программе, созданной на языке C++, это различие нельзя игнорировать. Если вы собираетесь выполнять над числом арифметические операции, вам требуется именно число, а не строка, то есть 1234, а не "1234". Если же вам нужно добавить запятую, чтобы отделить тысячи от сотен, тогда требуется строка "1234", которую можно превратить в "1,234". Когда программа должна вводить числа, часто удобнее считывать их как строки символов, редактировать в таком виде и затем преобразовывать в числа. Скажем, если программа должна прочитать значение денежной суммы, пользователь может ввести перед ним символ доллара, а может и не вводить, а когда программа считывает процентное значение, можно поставить в конце знак процента, а можно и не ставить. Программе эти знаки не нужны. Поэтому она может прочитать введенное значение как строку символов, сохранить его в строковой переменной C и удалить ненужные символы, оставив только цифры. После этого строку нужно будет преобразовать в число, что очень легко выполнить с помощью стандартной функции `atoi`.

Эта функция принимает один аргумент — строку C — и возвращает значение типа `int`, соответствующее представленному этой строкой числу. Так, `atoi("1234")` возвращает число 1234. Если строка не содержит числа, функция возвращает `0`. Например, `("#37")` возвращает `0`, поскольку значение аргумента `("#37")` не является цифрой. Название этой функции представляет собой сокращение от англ. alphabetic to integer (произносится как «эй-ту-ай»). Функция `atoi` определена в библиотеке с заголовочным файлом `<cstdlib>`, так что любая программа, в которой она используется, должна содержать следующую директиву:

```
#include <cstdlib>
```

Если число слишком велико для преобразования его к типу данных `int`, можно преобразовать строку в значение типа `long`. Это выполняет функция `atol`, точно такая же, как `atoi`, и отличающаяся от нее лишь типом возвращаемого значения.

В листинге 11.1 приведено определение функции `read_and_clean`, считывающей введенную пользователем строку и удаляющей из нее все символы, кроме цифр

от '0' до '9'. С помощью функции atoi она преобразует «очищенную» строку в целочисленное значение. Как показывает демонстрационная программа, функцию read_and_clean можно использовать для ввода значений денежных сумм независимо от того, добавит ли пользователь символ доллара. Точно так же с ее помощью можно вводить процентные значения независимо от наличия знака процента. Посмотрев на выходные данные, можно подумать, что функция просто удаляет из входной строки некоторые символы, но на самом деле она делает больше: формирует значение типа int, которым можно пользоваться в программе как числом, а не как строкой символов.

Листинг 11.1. Преобразование строки C в целое число

```
// Демонстрирует функцию read_and_clean.
#include <iostream>
#include <cstdlib>
#include <cctype>

void read_and_clean(int& n);
// Считывает входную строку. Удаляет все символы, кроме цифр.
// Преобразует полученную строку C в число и записывает
// его в параметр n.

void new_line();
// Удаляет из входного потока все символы до конца текущей строки.
// Удаляет также символ '\n' в конце строки.

int main()
{
    using namespace std;
    int n;
    char ans;
    do
    {
        cout << "Enter an integer and press Enter: ";
        read_and_clean(n);
        cout << "That string converts to the integer "
             << n << endl;
        cout << "Again? (yes/no): ";
        cin >> ans;
        new_line();
    } while ( (ans != 'n') && (ans != 'N') );
    return 0;
}

// Используем библиотеки классов iostream, cstdlib и cctype.
void read_and_clean(int& n)
{
    using namespace std;
    const int ARRAY_SIZE = 6;
    char digit_string[ARRAY_SIZE];

    char next;
    cin.get(next);
    int index = 0;
    while (next != '\n')
```

```
    {
        if ( (isdigit(next)) && (index < ARRAY_SIZE - 1) )
        {
            digit_string[index] = next;
            index++;
        }
        cin.get(next);
    }
    digit_string[index] = '\0';

n = atoi(digit_string);
}

// Используем библиотеку классов iostream.
void new_line()
{
    using namespace std;

... Остальная часть определения функции new_line приведена в листинге 5.6. ...

}
```

Пример диалога

```
Enter an integer and press Enter: $ 100
That string converts to the integer 100
Again? (yes/no): yes
Enter an integer and press Enter: 100
That string converts to the integer 100
Again? (yes/no): yes
Enter an integer and press Enter: 99%
That string converts to the integer 99
Again? (yes/no): yes
Enter an integer and press Enter: 23% &&5 *12
That string converts to the integer 23512
Again? (yes/no): no
```

Функция read_and_clean, приведенная в листинге 11.1, удаляет из введенной пользователем строки все нецифровые символы, но она не может проверить, соответствует ли оставшаяся последовательность символов тому числу, которое хотел ввести пользователь. Нужно предоставить ему возможность посмотреть на результат преобразования и подтвердить, что все сделано правильно. Если получено неверное значение, пользователь должен иметь возможность повторить ввод. В листинге 11.2 функция read_and_clean вызывается из другой функции, называющейся get_int, которая принимает все данные, введенные пользователем, и позволяет ему повторять ввод, пока он не будет удовлетворен числом, полученным из введенной им строки. Это довольно надежная процедура ввода. (Функция get_int представляет собой усовершенствованную версию одноименной функции, приведенной в листинге 5.6.)

Функции read_and_clean из листинга 11.1 и get_int из листинга 11.2 являются примерами пользовательских функций для ввода числовых значений. В практическом задании 3 к этой главе вам предлагается определить функцию, похожую на get_int, считывающую число типа double, а не int. Для этого была бы полезна

стандартная функция, преобразующая строковое значение в число типа double. Такая функция есть — она называется atof и находится в той же библиотеке классов с заголовочным файлом cstdlib. Например, функция atof("9.99") возвращает значение 9.99 типа double. Если строка не содержит число, которое можно интерпретировать как значение типа double, то данная функция возвращает 0.0. Название функции представляет собой сокращение от англ. alphabetic to floating point (произносится как «эй-ту-эф»). Вспомните, что числа с десятичной точкой часто называют числами с плавающей запятой из-за особого способа их хранения в памяти компьютера.

Преобразование строки C в число

Функции atoi, atol и atof можно использовать для преобразования строки C, состоящей из цифр, в числовое значение. Первые две функции преобразуют строку C в целочисленное значение. Единственная разница между ними заключается в том, что atoi возвращает значение типа int, а atol — значение типа long. Третья функция — atof — преобразует строку C в значение типа double. Если строка C, переданная в качестве аргумента любой из этих функций, не может быть преобразована в число, то функция возвращает 0. Например, оператор

```
int x = atoi("657");
```

присваивает переменной x значение 657, а оператор

```
double y = atof("12.37");
```

присваивает переменной y значение 12.37.

Любая программа, использующая функцию atoi, atol или atof, должна содержать следующую директиву:

```
#include <cstdlib>
```

Листинг 11.2. Функция ввода с повышенной надежностью

```
// Демонстрационная программа для усовершенствованной версии
// функции get_int.
#include <iostream>
#include <cstdlib>
#include <cctype>

void read_and_clean(int& n);
// Считывает входную строку. Удаляет все символы, кроме цифр. Преобразует
// результирующую строку C в число и записывает его в параметр n.

void new_line();
// Удаляет из входного потока все символы до конца текущей строки.
// Удаляет также символ '\n' в конце строки.

void get_int(int& input_number);
// Присваивает параметру input_number число, правильность которого
// подтвердит пользователь.

int main()
```

```
{
    using namespace std;
    int input_number;
    get_int(input_number);
    cout << "Final value read in = " << input_number << endl;
    return 0;
}

// Используем библиотеку классов iostream и функцию read_and_clean.
void get_int(int& input_number)
{
    using namespace std;
    char ans;
    do
    {
        cout << "Enter input number: ";
        read_and_clean(input_number);
        cout << "You entered " << input_number
             << " Is that correct? (yes/no): ";
        cin >> ans;
        new_line();
    } while ((ans != 'y') && (ans != 'Y'));
}


// Используем библиотеки классов iostream, cstdlib и cctype.
void read_and_clean(int& n)
{

... Определение функции read_and_clean приведено в листинге 11.1. ...

}

// Используем библиотеку классов iostream.
void new_line()
{

... Определение функции new_line приведено в листинге 5.6. ...

}
```

Пример диалога

```
Enter input number: $57
You entered 57 Is that correct? (yes/no): no
Enter input number: $77*5xa
You entered 775 Is that correct? (yes/no): no
Enter input number: 77
You entered 77 Is that correct? (yes/no): no
Enter input number: $75
You entered 75 Is that correct? (yes/no): yes
Final value read in = 75
```

11.2. Стандартный класс string

Ловлю себя и вас на каждой фразе, на каждом слове и спешу скорее запереть все эти фразы и слова в свою литературную кладовую: авось пригодятся!

Антон Чехов

В разделе 11.1 вы познакомились со строками С и узнали, что это просто массивы символов, в которых конец строки отмечается символом `'\0'`. При работе с такими строками необходимо учитывать их структуру. Скажем, если в конец строки нужно добавить символы, а в массиве для них недостаточно места, приходится создавать новый массив, вмещающий строки большей длины. Короче говоря, работая со строками С, нужно помнить об особенностях их реализации и хранения в памяти. Это является источником лишней работы и часто приводит к ошибкам. Стандарт C++ ANSI/ISO определяет, что в языке должен быть задан класс `string`, позволяющий работать со строками как с базовым типом данных и не заботиться о деталях его реализации. С этим типом данных вы сейчас и познакомитесь.

Первое знакомство с классом string

Класс `string` определен в библиотеке с именем `<string>` и отнесен к пространству имен `std`. Поэтому для его использования в программу нужно включить следующие директивы:

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

Класс `string` позволяет работать со значениями и выражениями типа `string` почти так же, как со значениями простого типа данных. Так, для присваивания значения переменной типа `string` можно пользоваться оператором `=`, а для конкатенации двух строк — оператором `+`. Предположим, что `s1`, `s2` и `s3` являются объектами типа `string` и переменные `s1` и `s2` содержат строковые значения. Тогда переменной `s3` можно присвоить строку, состоящую из `s1`, в конец которой добавлена строка `s2`, при помощи оператора:

```
s3 = s1 + s2;
```

выполняющего присваивание и конкатенацию.

И при этом нет риска, что `s3` окажется слишком маленькой для нового значения. Если сумма значений длины строк `s1` и `s2` превысит вместимость переменной `s3`, для нее будет автоматически выделена дополнительная память.

Как отмечалось ранее в этой главе, заключенные в двойные кавычки строковые литералы в программе на C++ фактически являются строками С, и поэтому они не относятся к типу `string`. Но ими можно пользоваться как литеральными значениями типа `string`, и мы часто будем говорить о них как о значениях типа `string`. Так, оператор

```
s3 = "Hello Mom!";
```

присваивает переменной `s3` объект `string`, содержащий те же символы и в том же порядке, что и строка С `"Hello Mom!"`.

У класса string имеется используемый по умолчанию конструктор, инициализирующий объект типа string пустой строкой. Кроме того, у него есть второй конструктор с одним аргументом, в котором задается стандартная строка С. Этот второй конструктор инициализирует объект типа string значением, представляющим ту же строку, что и аргумент. В качестве примера рассмотрим следующие две строки:

```
string phrase;
string noun("ants");
```

В первой из них объявляется строковая переменная phrase, инициализируемая пустой строкой. Во второй объявляется строковая переменная noun, которая инициализируется строкой, подобной строке С "ants". Большинство программистов скажут, что переменная noun инициализируется строкой "ants", но это неверно. На самом деле здесь имеет место приведение типов. Заключенная в кавычки строка "ants" является строкой С, а не значением типа string. Переменной noun присваивается объект типа string, содержащий те же символы и в том же порядке, что и строка "ants", но его значение не завершается нуль-символом '\0'. И теоретически программисту вообще не нужно знать, хранятся в нем строки в виде массивов или в виде какой-нибудь другой структуры данных.

Существует альтернативный синтаксис объявления переменной типа string и вызова конструктора этого типа. Приведенные ниже две строки:

```
string noun("ants");
string noun = "ants";
```

являются эквивалентными.

Все эти особенности класса string демонстрирует маленькая программа, приведенная в листинге 11.3. Обратите внимание, что значения типа string можно выводить с помощью оператора <<.

Листинг 11.3. Программа с использованием класса string

```
// Демонстрирует стандартный класс string.
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string phrase; // Инициализируется пустой строкой.
    // Два способа инициализации строковой переменной.
    string adjective("fried"), noun("ants");
    string wish = "Bon appetite!";

    phrase = "I love " + adjective + " " + noun + "!";
    cout << phrase << endl
         << wish << endl;

    return 0;
}
```

Пример диалога

```
I love fried ants!
Bon appetite!
```

Рассмотрим следующую строку из программы листинга 11.3:

```
phrase = "I love " + adjective + " " + noun + "!";
```

Для того чтобы программист мог выполнять конкатенацию строк таким простым и естественным способом, разработчикам языка C++ пришлось немало потрудиться. Строковая константа "I love " не является объектом типа string. Она хранится как строка C (в виде массива символов с нуль-терминатором). Встретив строку "I love " в качестве аргумента оператора +, C++ находит его перегруженное определение для таких значений как "I love ". Существует несколько подобных перегруженных версий этого оператора: со строкой C слева и объектом string справа, со строкой C справа и объектом типа string слева, со строками C с обеих сторон (также возвращающая объект string). И конечно, есть версия с двумя объектами типа string.

На самом деле наличие такого количества перегруженных версий оператора конкатенации в языке не обязательно. Если бы их не было, C++ просто вызывал бы конструктор класса string, преобразующий строку C "I love " в объект типа string, к которому можно применить оператор +. Это преобразование выполняет конструктор с одним строковым параметром. Но наличие полного набора перегруженных версий оператора + позволяет выполнять конкатенацию более эффективно.

Класс string

Класс string может использоваться для представления строк символов, причем более гибкого и эффективного, чем строки C, о которых рассказывалось в разделе 11.1.

Данный класс определен в библиотеке string, и это определение отнесено к пространству имен std. Поэтому программы, в которых он применяется, должны содержать следующие (или эквивалентные им) директивы:

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

У класса string имеется используемый по умолчанию конструктор, инициализирующий объект типа string пустой строкой, и конструктор, который принимает в качестве аргумента строку C и инициализирует объект типа string представленным ею значением. Например:

```
string s1, s2("Hello");
```

После появления класса string некоторые программисты считают, что следует использовать только его. Однако на самом деле обойтись без строк C в программах на C++ непросто.

Ввод-вывод с помощью класса string

Для вывода объектов типа string можно пользоваться оператором <<, точно так же, как для вывода данных других типов. Эту простую операцию демонстрирует программа, приведенная в листинге 11.3. А вот в отношении ввода данных класса такого типа имеется одна тонкость.

Оператор ввода >> для объектов типа string работает так же, как для других данных, но помните, что он игнорирует начальные пробелы и прекращает чтение, как

только встретит символ пробела, табуляции или перевода строки. В качестве примера рассмотрим следующий код:

```
string s1, s2;
cin >> s1;
cin >> s2;
```

Если пользователь введет

```
May the hair on your toes grow long and curly!
```

объекту s1 будет присвоена строка "May" без ведущих и завершающих пробелов, а переменой s2 будет присвоена строка "the". То есть, оператор ввода позволяет считывать только слова, но не строки, содержащие пробелы. Иногда именно это и требуется, но в общем случае считывания одних лишь слов недостаточно.

Если нужно, чтобы программа прочитала всю введенную пользователем строку в переменную типа string, можно воспользоваться функцией getline. Синтаксис ее использования со строковыми объектами несколько отличается от синтаксиса, описанного в разделе 11.1 для строк С. В частности, для ввода с клавиатуры не применяется вызов cin.getline, вместо этого вызывается функция getline (не являющаяся функцией-членом объекта cin) и объект cin передается ей в качестве аргумента.

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line);
cout << line << "END OF OUTPUT\n";
```

Включив в состав завершенной программы этот код, можно получить следующий диалог:

```
Enter some input:
Do be do to you!
Do be do to you!END OF OUTPUT
```

Если в строке имеются ведущие или завершающие пробелы, они также становятся частью строкового значения, прочитанного функцией getline. Данная версия getline находится в библиотеке string. Вместо cin ей можно передавать потоковый объект, подключенный к текстовому файлу.

Ввод-вывод данных объектов типа string

Для вывода данных в объект типа string можно пользоваться оператором << и объектом cout, а для ввода — оператором >> и объектом cin. Однако при использовании оператора >> для ввода данных в объект типа string считывается только часть строки, ограниченная символами пробела, табуляции или перевода строки. Чтобы ввести всю строку с клавиатуры, можно применять функцию getline.

Примеры

```
string greeting("Hello"), response, next_word;
cout << greeting << endl;
getline(cin, response);
cin >> next_word;
```

Для чтения пустого символа нельзя использовать cin и >>. Посимвольное чтение строки можно выполнить с помощью функции cin.get, описанной в главе 5. Она считывает значения типа char, а не string, но может быть полезна для ввода данных в объект string. В листинге 11.4 приведена программа, демонстрирующая использование функций getline и cin.get для ввода данных в объект типа string. Назначение функции new_line объясняется в разделе «Ловушка: одновременное использование потока cin с оператором >> и функции getline» этой главы.

Листинг 11.4. Программа, использующая класс string

```
// Демонстрирует использование функций getline и cin.get.
#include <iostream>
#include <string>

void new_line();

int main()
{
    using namespace std;

    string first_name, last_name, record_name;
    string motto = "Your records are our records.";

    cout << "Enter your first and last name:\n";
    cin >> first_name >> last_name;
    new_line();

    record_name = last_name + ", " + first_name;
    cout << "Your name in our records is: ";
    cout << record_name << endl;

    cout << "Our motto is\n"
         << motto << endl;
    cout << "Please suggest a better (one line) motto:\n";
    getline(cin, motto);
    cout << "Our new motto will be:\n";
    cout << motto << endl;

    return 0;
}

// Используем библиотеку классов iostream.
void new_line()
{
    using namespace std;

    char next_char;
    do
    {
        cin.get(next_char);
    } while (next_char != '\n');
}
```

Пример диалога

```
Enter your first and last name:
  B'Elanna Torres
```

```
Your name in our records is: Torres, B'Elanna
Our motto is
Your records are our records.
Please suggest a better (one-line) motto:
Our records go where no records dared to go before.
Our new motto will be:
Our records go where no records dared to go before.
```

Упражнения для самопроверки

15. Рассмотрим код, входящий в состав правильно написанной программы:

```
string s1, s2;
cout << "Enter a line of input:\n";
cin >> s1 >> s2;
cout << s1 << "*" << s2 << "<END OF OUTPUT";
```

Если диалог начинается указанным образом:

```
Enter a line of input:
A string is a joy forever!
```

какой будет следующая выведенная программой строка?

16. Рассмотрим код, входящий в состав правильно написанной программы:

```
string s;
cout << "Enter a line of input:\n";
getline(cin, s);
cout << s << "<END OF OUTPUT";
```

Если диалог начинается указанным образом:

```
Enter a line of input:
A string is a joy forever!
```

какой будет следующая выведенная программой строка?

Совет программисту: дополнительные версии функции getline

До сих пор в данной главе описывался такой способ вызова функции `getline`:

```
string line;
cout << "Enter a line of input:\n";
getline(cin, line);
```

Эта версия функции прекращает чтение, встретив символ конца строки `'\n'`. Еще одна версия названной функции позволяет задать другой сигнальный символ, отмечающий конец входных данных. Например, вызов функции `getline` в приведенном фрагменте кода:

```
string line;
cout << "Enter some input:\n";
getline(cin, line, '?');
```

указывает, что чтение данных будет прекращено, когда во входном потоке встретится знак вопроса.

Функцию getline можно использовать как функцию типа void, но на самом деле она возвращает ссылку на свой первый аргумент, которым в приведенном фрагменте кода является cin. Поэтому при выполнении кода

```
string s1, s2;
getline(cin, s1) >> s2;
```

в переменную s1 будет прочитана строка текста, а в переменную s2 — следующее за этой строкой слово (строка символов без пробелов).

Вызов getline(cin, s1) возвращает ссылку на cin, так что после него выполняется оператор

```
cin >> s2;
```

Этот способ использования функции getline скорее необычен, нежели полезен, но все же иногда и он может пригодиться.

Ловушка: комбинирование ввода с помощью потока cin и функции getline

Применение программного кода, в котором одновременно используются поток cin с оператором >> для ввода значений переменных и функция getline, требует особой аккуратности. В качестве примера рассмотрим фрагмент кода:

```
int n;
string line;
cin >> n;
getline(cin, line);
```

Когда при его выполнении считываются такие входные данные:

```
42
Hello hitchhiker.
```

можно предположить, что переменной n присваивается значение 42, а переменной line — значение, представляющее строку "Hello hitchhiker.".

Но на самом деле переменной n действительно присваивается значение 42, а вот переменной line — пустая строка. В чем же дело?

Оператор cin >> n; пропускает ведущие пробелы во входном потоке, но сохраняет для следующего оператора ввода остаток строки, в данном случае символ '\n'. Он всегда оставляет следующему вызову функции getline остаток строки (хотя бы только названный символ). В данном случае эта функция первым видит символ '\n' и тут же прекращает чтение, так что в результате считывается пустая строка. Если вам покажется, что программа непонятным образом игнорирует входные данные, посмотрите, не вызывается ли функция getline вслед за оператором >>. Во избежание такой накладки можно пользоваться функцией new_line, приведенной в листинге 11.4, или функцией ignore из библиотеки iostream. Например:

```
cin.ignore(1000, '\n');
```

При таких аргументах функция-член ignore удалит из входного потока весь остаток строки, включая символ '\n' (или 1000 символов, если среди них не окажется ...ола '\n').

Использование функции getline с объектами типа string

У функции getline есть две версии для работы с объектами типа string:

```
istream& getline(istream& ins, string& str_var, char delimiter);
```

и

```
istream& getline(istream& ins, string& str_var);
```

Первая из них считывает символы из объекта istream, переданного в качестве первого аргумента (во всех примерах этой главы — объекта cin), и помещает эти символы в переменную str_var типа string до тех пор, пока не встретит символ delimiter. Данный символ удаляется из входного потока, но в переменную не помещается. Во второй версии в качестве значения delimiter по умолчанию используется символ '\n'. В остальном обе версии работают одинаково.

Функция getline возвращает свой первый аргумент (во всех примерах этой главы — объект cin), но обычно применяется как функция типа void.

Одновременное использование оператора >> и функции getline может вызвать также другие проблемы. Особенно часто они встречаются при переходе от одного компилятора к другому. В подобных случаях, в частности когда от программы требуется высокая степень переносимости, можно попробовать выполнять посимвольный ввод с помощью функции cin.get.

Обработка строк с помощью класса string

Класс string позволяет выполнять над строковыми значениями все те операции, о которых рассказывалось в разделе 11.1, посвященном строкам C, и даже больше.

Данный класс поддерживает доступ к символам как к элементам массива, так что объекты этого класса обладают всеми преимуществами массивов символов плюс дополнительными преимуществами, которых нет у массивов, например возможностью автоматического увеличения их вместимости.

Если объект типа string назван last_name, то доступ к *i*-му символу представляемой им строки выполняется с помощью обращения last_name[i]. Использование такого синтаксиса доступа показано в программе из листинга 11.5.

Здесь же демонстрируется применение функции-члена length, имеющейся у каждого объекта типа string, — она возвращает длину представляемой им строки. Таким образом, объект типа string может использоваться не только как обычный массив, но и как частично заполненный массив с автоматическим отслеживанием количества заполненных элементов.

Листинг 11.5. Доступ к объекту типа string как к массиву

```
// Демонстрирует использование объекта типа string как массива.
#include <iostream>
#include <string>
using namespace std;

int main()
{
```

продолжение ⇩

Листинг 11.5 (*продолжение*)

```
    string first_name, last_name;

    cout << "Enter your first and last name:\n";
    cin >> first_name >> last_name;

    cout << "Your last name is spelled:\n";
    int i;
    for (i = 0; i < last_name.length(); i++)
    {
        cout << last_name[i] << " ";
        last_name[i] = '-';
    }
    cout << endl;
    for (i = 0; i < last_name.length(); i++)
        cout << last_name[i] << " "; // Выводит "-" под каждой буквой.
    cout << endl;

    cout << "Good day " << first_name << endl;
    return 0;
}
```

Пример диалога

```
Enter your first and last name:
John Crichton
Your last name is spelled:
C r i c h t o n
- - - - - - - -
Good day John
```

Когда для объекта типа `string` задается индекс в квадратных скобках, этот объект не проверяет допустимость заданного индекса. И если индекс оказывается недопустимым (то есть большим или равным длине строки, хранящейся в объекте), последствия непредсказуемы — программа может выполнить ошибочные действия, не выводя сообщения об ошибке.

Однако у объектов `string` имеется функция-член `at`, проверяющая заданный индекс. Она выполняет ту же задачу, что и квадратные скобки, но отличается от них, во-первых, синтаксисом (вместо вызова `a[i]` используется вызов `a.at(i)`), а во-вторых, тем, что она проверяет, допустим ли индекс, переданный ей в качестве аргумента. И если значение `i` в вызове `a.at(i)` оказывается недопустимым, функция выводит сообщение с указанием ошибки, содержащейся в программе. В следующих двух фрагментах кода выполняются попытки доступа к символам за пределами строки, первая из этих попыток проходит незамеченной (то есть без сообщения об ошибке):

```
string str("Mary");
cout << str[6] << endl;
```

а вторая:

```
string str("Mary");
cout << str.at(6) << endl;
```

вызывает немедленное завершение программы с указанием ошибки. Но учтите, что в некоторых системах сообщения об этой ошибке довольно невразумительные.

Объект типа string позволяет изменить один символ строки путем присвоения значения типа char индексированной переменной, такой как str[i]. То же самое можно сделать и с помощью функции-члена at. Например, для замены третьего символа объекта str типа string символом 'X' можно выполнить один из двух операторов:

```
str.at(2)='X';
```

или

```
str[2]='X';
```

Как и в обычном символьном массиве, позиции символов в объекте string нумеруются с нуля, поэтому третий символ строки имеет индекс 2.

Список часто используемых функций-членов класса string приведен в табл. 11.2.

Многие операции с объектами этого класса производить удобнее, чем со строками С, описанными в разделе 11.1. В частности, оператор ==, выполняемый над объектами класса string, возвращает результат, который соответствует нашему традиционному пониманию равенства строк — то есть значение true, если две строки содержат одинаковые символы в одинаковом порядке, и значение false в противном случае. Операторы сравнения <, >, <=, >= сравнивают строки с точки зрения их лексикографического порядка. (Лексикографический (словарный) порядок — это обычный алфавитный порядок сортировки на основе набора символов ASCII, приведенного в приложении 3. Если строки состоят лишь из букв, причем либо только строчных, либо только прописных, лексикографический порядок полностью совпадает с обычным алфавитным порядком сортировки.)

Операторы сравнения для объектов типа string

При использовании со стандартным типом string языка C++ операторы ==, !=, <, >, <=, >= соответствуют нашему традиционному представлению о сравнении строк. Эти операторы ведут себя не так, как со строками С (см. раздел 11.1).

Таблица 11.2. Функции-члены стандартного класса string

Пример	Примечания
Конструкторы	
string *строка*;	Используемый по умолчанию конструктор создает пустой объект *строка* типа string
string *строка*("*образец*");	Создает объект типа string с данными "*образец*"
string *строка*(*строковая_переменная*);	Создает объект типа string с именем *строка*, представляющий собой копию объекта *строковая_переменная* (*строковая_переменная* — объект класса string)
Доступ к элементам	
строка[i]	Возвращает ссылку на символ строки *строка* с индексом i для чтения-записи. Не проверяет допустимость индекса

продолжение

Таблица 11.2 (*продолжение*)

Пример	Примечания
строка.at(i)	Возвращает ссылку на символ строки с индексом i для чтения-записи. Проверяет допустимость индекса, а в остальном эквивалентна выражению *строка*[i]
строка.substr(*позиция*, *длина*)	Возвращает подстроку из вызывающего объекта, начиная с индекса *позиция*, длиной *длина* символов
Присваивание/модификация	
строка1 = *строка2*;	Инициализирует объект *строка1* данными объекта *строка2*
строка1 += *строка2*;	Добавляет строку объекта *строка2* в конец строки объекта *строка1*
строка.empty()	Возвращает true, если *строка* содержит пустую строку, и false в противном случае
строка1 + *строка2*	Возвращает строку, представляющую собой конкатенацию строк *строка1* и *строка2*
строка.insert(*позиция*, *строка2*);	Вставляет строку *строка2* в строку *строка*, начиная с позиции *позиция*
строка.remove(*позиция*, *длина*);	Удаляет подстроку длиной *длина*, начиная с позиции *позиция*
Сравнение	
строка1 == *строка2* *строка1* != *строка2*	Сравнение на равенство и на неравенство. Возвращает значение типа bool
строка1 <= *строка2* *строка1* >= *строка2*	Четыре сравнения. Выполняются на основе лексикографического порядка
Поиск	
строка.find(*строка1*)	Возвращает индекс первого вхождения *строка1* в строку *строка*
строка.find(*строка1*, *позиция*)	Возвращает индекс первого вхождения *строка1* в строку *строка*; поиск начинается с позиции *позиция*
строка.find_first_of(*строка1*, *позиция*)	Возвращает индекс первого вхождения в строку *строка* любого символа строки *строка1*; поиск начинается с позиции *позиция*
строка.find_first_not_of(*строка1*, *позиция*)	Возвращает индекс первого вхождения в строку *строка* любого символа, отсутствующего в строке *строка1*; поиск начинается с позиции *позиция*

Пример: проверка палиндрома

Палиндром — это строка, одинаково читающаяся слева направо и справа налево. Программа, приведенная в листинге 11.6, проверяет входную строку, чтобы узнать, является ли она палиндромом. Эта программа игнорирует пробелы и знаки

препинания и считает прописную и строчную версии буквы одним и тем же символом. Вот несколько примеров палиндромов:

```
Able was I ere I saw Elba.
I Love Me, Vol. I.
Madam, I'm Adam.
A man, a plan, a canal, Panama.
Rats live on no evil star.
radar
deed
mom
racecar
```

Функция remove_punct интересна тем, что в ней используются функции-члены substr и find объекта string. Функция-член substr извлекает из вызывающего объекта подстроку заданной длины, начиная с указанной позиции. В трех первых строках функции remove_punct объявляются необходимые переменные. Затем выполняется цикл for, в котором перебираются все символы параметра s и производится поиск каждого из них в строке punct. Поиск выполняется посредством функции-члена find. Если искомый символ не будет найден в строке punct, он добавляется в конец результирующей строки no_punct.

Листинг 11.6. Программа для проверки палиндрома

```
// Проверяет, является ли введенная строка палиндромом.
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

void swap(char& v1, char& v2);
// Меняет местами значения переменных v1 и v2.

string reverse(const string& s);
// Возвращает копию s с символами, расположенными в обратном порядке.

string remove_punct(const string& s, const string& punct);
// Возвращает копию s, из которой удалены все вхождения символов
// строки punct.

string make_lower(const string& s);
// Возвращает копию s, в которой все буквы верхнего регистра
// переведены в нижний, а остальные символы остались без изменений.

bool is_pal(const string& s);
// Возвращает true, если строка является палиндромом,
// и false в противном случае.

int main()
{
    string str;
    cout << "Enter a candidate for palindrome test\n"
         << "followed by pressing Enter.\n";
```

продолжение

Листинг 11.6 (*продолжение*)

```
    getline(cin, str);

    if (is_pal(str))
        cout << "\"" << str + "\" is a palindrome.";
    else
        cout << "\"" << str + "\" is not a palindrome.";
    cout << endl;

    return 0;
}

void swap(char& v1, char& v2)
{
    char temp = v1;
    v1 = v2;
    v2 = temp;
}


string reverse(const string& s)
{
    int start = 0;
    int end = s.length();
    string temp(s);

    while (start < end)
    {
       end--;
       swap(temp[start], temp[end]);
       start++;
    }

    return temp;
}


// Используем библиотеки классов cctype и string.
string make_lower(const string& s)
{
    string temp(s);
    for (int i = 0; i < s.length(); i++)
        temp[i] = tolower(s[i]);

    return temp;
}


string remove_punct(const string& s, const string& punct)
{
    string no_punct; // Инициализируется пустой строкой.
    int s_length = s.length();
    int punct_length = punct.length();
```

```
    for (int i = 0; i < s_length; i++)
    {
        // Извлечение очередного символа из строки s.
        string a_char = s.substr(i,1);
        // Поиск вхождения очередного символа строки s в строке punct.
        int location = punct.find(a_char, 0);

      if (location < 0 || location >= punct_length)
        no_punct = no_punct + a_char; // a_char отсутствует
                                      // в punct, сохраняем его.
    }

    return no_punct;
}


// Используем функции make_lower, remove_punct.
bool is_pal(const string& s)
{
    string punct(",:;.?!'\" "); // Включает пробел.
    string str(s);
    str = make_lower(str);
    string lower_str = remove_punct(str, punct);

    return (lower_str == reverse(lower_str));
}
```

Примеры диалога

```
Enter a candidate for palindrome test
followed by pressing Enter.
Madam, I'm Adam.
"Madam, I'm Adam." is a palindrome.

Enter a candidate for palindrome test
followed by pressing Enter.
Radar
"Radar" is a palindrome.

Enter a candidate for palindrome test
followed by pressing Enter.
Am I a palindrome?
"Am I a palindrome?" is not a palindrome.
```

Упражнения для самопроверки

17. Рассмотрим следующий фрагмент кода:

```
    string s1, s2("Hello");
    cout << "Enter a line of input:\n";
    cin >> s1;
    if (s1 == s2)
        cout << "Equal\n";
    else
        cout << "Not equal\n";
```

Если диалог начинается таким образом:

```
Enter a line of input:
Hello friend!
```

какой будет следующая выведенная этим кодом строка?

18. Что должен выводить приведенный ниже код (включенный в состав правильно написанной программы на C++):

```
string s1, s2("Hello");
s1 = s2;
s2[0] = 'J';
cout << s1 << " " << s2;
```

Взаимное преобразование объектов типа string и строк C

Как вы знаете, C++ выполняет автоматическое преобразование типов, позволяющее присваивать строки C переменным типа string. Например, приведенный ниже код является абсолютно правильным.

```
char a_c_string[] = "This is my C string.";
string string_variable;
string_variable = a_c_string;
```

А вот такой оператор:

```
a_c_string = string_variable; // НЕДОПУСТИМО
```

вызовет вывод компилятором сообщения об ошибке.

Недопустим и следующий оператор:

```
strcpy(a_c_string, string_variable); // НЕДОПУСТИМО
```

Функция strcpy не может принимать в качестве второго аргумента объект типа string, и C++ не выполняет автоматического преобразования объектов string в строки C, что является реальной проблемой.

Для получения строки C, соответствующей объекту string, нужно выполнить явное преобразование типов. Это можно сделать с использованием функции-члена c_str(). Вот как правильно произвести копирование объекта string в строку C:

```
strcpy(a_c_string, string_variable.c_str()); // ДОПУСТИМО
```

Обратите внимание, что копирование нельзя выполнить без помощи функции strcpy. Функция-член c_str() возвращает строку C, соответствующую вызывающему объекту string. Как уже говорилось ранее в этой главе, оператор присваивания не работает со строками C. Поэтому если вы полагаете, что следующий оператор:

```
a_c_string = string_variable.c_str(); // НЕДОПУСТИМО
```

может выполнить копирование строки из переменной типа string в строку C, то это не так. И мы подчеркиваем, что такой оператор недопустим.

11.3. Векторы

— Ну и ладно, съем, — сказала Алиса. — Если я от него стану побольше, смогу достать ключ, а если стану еще меньше, пролезу под дверь. Будь что будет — в сад я все равно заберусь!

Льюис Кэрролл

Вектор — это тип данных похожий на массив, но все же имеющий некоторые существенные отличия. В C++ созданный программой массив обладает фиксированной длиной, которая не может ни увеличиваться, ни уменьшаться, а вектор служит тем же целям, что и массив, но его длина может динамически изменяться.

Для понимания материала, приведенного в этом разделе, не требуется изучать предыдущие разделы данной главы.

Основные понятия

Вектор, подобно массиву, имеет базовый тип и содержит набор значений этого типа. Однако синтаксис использования и объявления векторной переменной отличается от синтаксиса массивов.

Векторная переменная `v` базового типа `int` объявляется так:

```
vector<int> v;
```

Выражение `vector<int>` является не просто именем типа данных, это — имя класса. Аналогичные классы существуют и для других базовых типов данных. Таким образом, приведенное выше объявление создает объект `v` класса `vector<int>`. Используемый по умолчанию конструктор приведенного класса создает пустой вектор (то есть вектор с пустыми элементами).

Элементы вектора, как и элементы массива, индексируются с нуля. Для их чтения или изменения задается имя вектора и индекс в квадратных скобках. Например, следующий код изменяет значение *i*-го элемента вектора `v` и выводит новое значение на экран (`i` — переменная типа `int`):

```
v[i] = 42;
cout << "The answer is " << v[i];
```

Однако на использование этого синтаксиса налагается одно специфическое для векторов ограничение: выражение `v[i]` может применяться для изменения значения *i*-го элемента вектора, но не может использоваться для его инициализации. Изменить можно только тот элемент, которому уже присвоено значение. Для первого добавления в вектор элемента с указанным индексом обычно используется функция `push_back`.

Элементы добавляются в вектор в определенном порядке: сначала в позицию 0, затем в позицию 1, позицию 2 и т. д. Функция-член `push_back` добавляет элемент в следующую доступную позицию. Скажем, приведенный ниже код присваивает начальные значения элементам 0, 1 и 2 вектора `sample`.

```
vector<double> sample;
sample.push_back(0.0);
```

```
sample.push_back(1.1);
sample.push_back(2.2);
```

Количество элементов вектора называется его *размером*. Для того чтобы узнать размер вектора, можно воспользоваться функцией-членом size. Так, после выполнения приведенного выше кода вызов sample.size() вернет значение 3. Вывод на экран всех элементов вектора sample можно выполнить так:

```
for (int i = 0; i < sample.size(); i++)
    cout << sample[i] << endl;
```

Функция size возвращает значение типа unsigned int, а не int. (Этот тип данных допускает использование только неотрицательных целых чисел.) Когда требуется значение типа int, возвращенное функцией size значение должно автоматически приводиться к данному типу, однако некоторые компиляторы выдают предупреждение о том, что вы пользуетесь типом unsigned int там, где требуется тип данных int. Если вам нужен абсолютно надежный код, можно выполнить явное приведение значения типа unsigned int к типу int или же использовать переменную типа unsigned int, как в следующем цикле for:

```
for (unsigned int i = 0; i < sample.size(); i++)
cout << sample[i] << endl;
```

Листинг 11.7 содержит простую программу, в которой демонстрируются некоторые базовые технологии работы с векторами.

Листинг 11.7. Использование вектора

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

int main()
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size() = " << v.size() << endl;
        cin >> next;
    }

    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

Пример диалога

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size() = 1
4 added. v.size() = 2
6 added. v.size() = 3
8 added. v.size() = 4
You entered:
2 4 6 8
```

У класса vector<int> имеется конструктор с одним целочисленным аргументом, инициализирующий заданное количество элементов вектора. Например, если объявить вектор v следующим образом:

```
vector<int> v(10);
```

его первые десять элементов будут инициализированы нулями, и вызов v.size() вернет 10. Тогда с помощью обращения v[i] вы сможете присваивать значения любым из первых десяти элементов вектора (то есть i может иметь значения от 0 до 9). В частности, непосредственно после приведенного выше объявления может располагаться такой код:

```
for (unsigned int i = 0; i < 10; i++)
    v[i] = i;
```

Для установки значений остальных элементов, индексы которых больше 9, придется все равно применять функцию push_back.

Векторы

Векторы похожи на массивы, но их размер не фиксирован. При добавлении очередного элемента размер вектора автоматически увеличивается. Векторы определены в библиотеке vector и отнесены к пространству имен std. Поэтому программа, в которой используются векторы, должна содержать следующий (или подобный) код:

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

Класс векторов для заданного *базового_типа* называется vector<*базовый_тип*>. Вот два простых примера объявления векторов:

```
vector<int> v; // С помощью используемого по умолчанию
               // конструктора создается пустой вектор.
vector<AClass> record(20); // Конструктор вектора вызывает используемый
                           // по умолчанию конструктор класса
                           // AClass для инициализации 20 элементов.
```

Элементы добавляются в вектор с помощью функции-члена push_back, как в следующем примере:

```
v.push_back(42);
```

После того как заданному элементу вектора присваивается первое значение, к нему можно обращаться как к элементу массива, задавая в квадратных скобках имя вектора и индекс.

При использовании конструктора с целочисленным аргументом векторы чисел инициализируются нулевыми значениями базового числового типа. Если базовым типом вектора является класс, инициализация выполняется применяемым по умолчанию конструктором этого класса.

Определение класса вектора находится в библиотеке `vector` и относится к пространству имен `std`. Поэтому программа, в которой используются векторы, должна содержать следующий (или подобный) код:

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

Ловушка: индекс в квадратных скобках превышает размер вектора

Если `v` — вектор, а значение `i` больше или равно `v.size()`, то элемента `v[i]` не существует, и прежде чем обращаться к нему с помощью выражения `v[i]`, нужно создать этот элемент посредством функции `push_back`. Например, если попытаться присвоить такому элементу новое значение с помощью следующего оператора:

```
v[i] = n;
```

можно не получить сообщения об ошибке, но с этого момента поведение программы будет непредсказуемым.

Совет программисту: учитывайте особенности присваивания для векторов

Оператор присваивания для векторов выполняет поэлементное присваивание значений их элементов (по мере необходимости увеличивая или уменьшая размер операнда, расположенного слева). Таким образом, если оператор присваивания базового типа данных создает независимую копию значения этого типа данных, оператор присваивания вектора создает независимую копию вектора.

Обратите внимание, что, создавая независимую копию вектора, заданного справа от него, оператор присваивания вектора полностью полагается на оператор присваивания базового типа данных. (О перегрузке оператора присваивания для классов подробно рассказывается в главе 12.)

Эффективное использование памяти

В любой момент времени вектор имеет емкость, то есть количество элементов, для которых на этот момент выделена память. Его емкость можно узнать с помощью функции-члена `capacity()`. Не путайте емкость вектора и его размер. *Размер* представляет количество элементов в заполненной части вектора, а *емкость* — это количество элементов, для которых выделена память. Емкость обычно больше размера, и никогда не бывает меньше.

Когда емкость вектора исчерпана и для него требуется дополнительная память, его емкость автоматически увеличивается. Насколько — зависит от конкретной

реализации, но всегда на большую величину, чем нужно в настоящий момент (как правило, она удваивается). Поскольку увеличение емкости является достаточно сложной задачей, выделение памяти большими блоками эффективнее, чем маленькими.

Размер и емкость

Размер вектора — это количество его заполненных элементов, а *емкость* вектора — количество элементов, для которых выделена память. Для вектора v размер и емкость можно определить с помощью функций-членов `v.size()` и `v.capacity()`.

Программист может не заботиться о емкости вектора, поскольку C++ управляет ею автоматически. Но при желании, когда важна эффективность работы программы, вы можете управлять ею самостоятельно с помощью функций-членов вектора. Посредством функции-члена `reserve` можно явно увеличить емкость вектора. Например, вызов

```
v.reserve(32);
```

устанавливает емкость вектора v равной 32 элементам, если его текущая емкость меньше этого значения. В противном случае емкость не изменяется. А вызов

```
v.reserve(v.size() + 10);
```

увеличивает емкость вектора до v на 10 элементов по сравнению с текущей, независимо от величины последней. Вы можете рассчитывать на то, что функция `reserve` увеличит емкость вектора, но нет гарантии, что эта функция уменьшит его емкость, если аргумент окажется меньше текущей емкости.

Изменить размер вектора можно с помощью функции-члена `resize`. Так, следующий вызов устанавливает размер вектора равным 24 элементам:

```
v.resize(24);
```

Если его размер меньше 24, в него добавляются новые элементы так же, как это делается посредством конструктора с целочисленным аргументом. Если же его размер превышает 24, все элементы, следующие за первыми двадцатью четырьмя, будет удалены. При необходимости емкость автоматически увеличивается. С помощью функций `reserve` и `resize` можно уменьшать размер и емкость вектора, когда лишние элементы больше не нужны.

Упражнения для самопроверки

19. Правильно ли написана следующая программа:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v(10);
    int i;

    for (i = 0; i < v.size(); i++)
        v[i] = i;
```

```
    vector<int> copy;
    copy = v;
    v[0] = 42;
    for (i = 0; i < copy.size(); i++)
        cout << copy[i] << " ";
    cout << endl;
    return 0;
}
```

Если да, что она выводит?

20. Каково различие между размером и емкостью вектора?

Резюме

- Строковая переменная C – это то же самое, что массив символов, но используемый несколько иным способом. Конец строки, хранящейся в массиве, отмечается нуль-символом `'\n'`.
- Со строковыми переменными C обычно работают как с массивами, а не как с простыми переменными числовых и символьных типов. Так, строковой переменой C нельзя присвоить значение с помощью знака равенства. Кроме того, нельзя сравнивать значения двух строковых переменных C, используя оператор ==. Для выполнения этих операций имеются специальные функции.
- Стандарт ANSI/ISO определяет библиотеку `string` с полнофункциональным классом `string`, предназначенным для представления строк символов.
- Объекты класса `string` намного удобнее строк C. В частности, определенные для них операторы присваивания и равенства (= и ==) используются с объектами класса `string` самым естественным и понятным образом.
- Векторы – это аналог массивов, размер которых может уменьшаться и увеличиваться во время выполнения программы.

Ответы к упражнениям для самопроверки

1. Следующие два оператора:

```
char string_var[10] = "Hello";
char string_var[10] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

эквивалентны друг другу, но не эквивалентны никаким другим операторам.

Следующие два оператора:

```
char string_var[6] = "Hello";
char string_var[] = "Hello";
```

эквивалентны друг другу, но не эквивалентны никаким другим операторам.

Следующий оператор:

```
char string_var[10] = {'H', 'e', 'l', 'l', 'o'};
```

не эквивалентен никаким другим.

2. "DoBeDo to you"

3. Из объявления переменной string_var следует, что в ней имеется место только для шести символов (включая символ '\n'). Функция strcat не проверяет, имеется ли в данной переменной место для добавления указанных символов, и поэтому она запишет в память все символы строки " and Good-bye.", даже если часть этой памяти не будет принадлежать массиву string_var. Это означает, что изменится память, модификация которой не предусматривалась. Последствия такого действия непредсказуемы, но очевидно, что это не приведет ни к чему хорошему.

4. Если функция strlen еще не определена, это можно сделать так:

```
int strlen(const char str[])
// Предусловие: str содержит строковое значение, ограниченное
// символом '\0'.
// Возвращает количество символов в строке str (не считая '\0').
{
    int index = 0;
    while (str[index] != '\0')
        index++;
    return index;
}
```

5. Максимальное количество символов равно пяти, поскольку шестой необходим для хранения нуль-терминатора '\0'.

6. а) 1;

   б) 1;

   в) 5 (включая '\0');

   г) 2 (включая '\0');

   д) 6 (включая '\0').

7. Они не эквивалентны. Первое из них помещает в массив после символов 'a', 'b' и 'c' символ '\0'. Второе просто присваивает трем последовательным элементам массива символы 'a', 'b' и 'c' и вообще не помещает в этот массив символ '\0'.

8.
```
int index = 0;
while ( our_string[index] != '\0' )
{
    our_string[index] = 'X';
    index++;
}
```

9. а) Если строковая переменная C по какой-либо причине не содержит нуль-терминатора '\0', цикл может выйти за пределы памяти, выделенной для этой переменной, и изменить хранящуюся там информацию. Для защиты памяти за границей массива следует изменить условие while, как показано в пункте б);

   б) `while( our_string[index] != '\0' && index < SIZE )`

10. ```
#include <cstring>
// В этой библиотеке определена функция strcpy.
...
strcpy(a_string, "Hello");
```

11. ```
I did it my way!
```

12. Строка "good. I hope." не поместится в конце массива a_string. Последние ее символы будут записаны в память, не принадлежащую этому массиву.

13. ```
Enter some input:
The
   time is now.
The-timeEND OF OUTPUT
```

14. Полный диалог будет выглядеть так:

```
Enter a line of input:
May the hair on your toes grow long and curly.
May t<END OF OUTPUT
```

15. ```
A*string<END OF OUTPUT
```

16. ```
A string is a joy forever!<END OF OUTPUT
```

17. Полный диалог будет выглядеть так:

```
Enter a line of input:
Hello friend!
Equal
```

Помните, что объект cin прекращает чтение по достижении символа пробела, табуляции или перевода строки.

18. ```
Hello Jello
```

19. Программа написана правильно. Она выводит следующее:

```
0 1 2 3 4 5 6 7 8 9
```

Обратите внимание, что изменение вектора v не изменяет вектор copy. Присваивание

```
copy = v;
```

создает абсолютно независимую копию вектора v.

20. Размер – это число заполненных элементов вектора, а его емкость – количество элементов, для которых выделена память. Обычно емкость больше размера, и уж никак не может быть меньше.

Практические задания

1. Напишите программу, считывающую предложение длиной до 100 символов и выводящую его же с откорректированными расстояниями между словами и правильной расстановкой заглавных букв. Иными словами, все последовательности из двух или более пробелов должны быть заменены одним. Предложение должно начинаться с заглавной буквы и не содержать других заглавных букв. Не учитывайте правильное написание имен: если их первые

буквы будут переведены в нижний регистр — не страшно. Символ перевода строки программа должна интерпретировать как один пробел, так что эти символы и один или несколько пробелов необходимо заменить одним пробелом. Предполагается, что предложение оканчивается точкой и не содержит других точек. Например, при таких входных данных:

```
the    Answer to life, the Universe, and everything the
IS 42.
```

программа должна вывести:

```
The answer to life, the universe, and everything is 42.
```

2. Напишите программу, считывающую строку текста и выводящую число слов в этой строке, а также количество вхождений каждой буквы. Под словом понимается любая последовательность букв, ограниченная с каждой стороны пробелом, точкой, запятой либо символом начала или конца строки. Можно считать, что ввод состоит только из букв, пробелов, запятых и точек. При выводе количества букв в строке прописная и заглавная версии одной и той же буквы должны интерпретироваться как одна буква. Выведите буквы в алфавитном порядке и перечислите только те из них, которые содержатся во входной строке. Например, для входной строки

```
I say Hi.
```

программа должна вывести следующее:

```
3 words
1 a
1 h
2 i
1 s
1 y
```

3. Приведите определение функции, объявленной так:

```
void get_double(double& input_number);
// Постусловие: переменной input_number
// присвоено одобренное пользователем значение.
```

Включите это определение в отладочную программу.

Можно считать, что пользователь вводит числа в обычном наиболее часто используемом формате, скажем 23.789, не применяя научный формат. В качестве образца используйте определение функции get_int, приведенное в листинге 11.2, чтобы функция считывала ввод в виде последовательности символов, редактировала полученную символьную строку и преобразовывала результат в число типа double. Вам необходимо будет определить функцию, подобную read_and_clean, но более сложную, чем функция, приведенная в листинге 11.1, поскольку она должна работать еще и с десятичной точкой. Это довольно простое задание. Его более сложная версия должна позволять пользователю вводить число как в обычном, так и в научном (экспоненциальном) формате. При этом функция должна сама без помощи пользователя определять формат, в котором введено число.

4. Напишите программу, считывающую имя человека в таком формате: имя, второе имя или инициал, фамилия. Затем программа выводит имя в следующем формате:

   Фамилия, Имя Инициал.

   Например, для введенной строки

   Mary Average User

   программа должна вывести следующее:

   User, Mary A.

   А для ввода

   Mary A. User

   она должна вывести

   User, Mary A.

   Программа должна работать таким же образом и помещать точку после промежуточного инициала, даже если пользователь ее не ввел. Необходимо, чтобы она позволяла пользователю не вводить инициал или второе имя, и тогда в ответ на введенную строку

   Mary User

   она должна вывести

   User, Mary

   Если вы работаете со строками C, считайте, что длина каждой составляющей имени не превышает 20 символов. В качестве альтернативы можете воспользоваться классом `string`.

   Подсказка. Можно применять для ввода не одну, а три строковые переменные меньшего размера. Возможно, вы сочтете более удобным не пользоваться функцией `getline`.

5. Напишите программу, считывающую строку текста и заменяющую все четырехбуквенные слова словом "love". Например, для входной строки

   I hate you, you dodo!

   программа должна вывести следующее:

   I love you, you love!

   Конечно, результат часто будет получаться бессмысленным. Так, для входной строки

   John will run home.

   будет выведено:

   Love love run love.

   Если четырехбуквенное слово начинается с заглавной буквы, оно должно быть заменено словом "Love", а не "love". Регистр остальных букв, кроме первой, проверять не нужно. Словом называется любая строка, состоящая из букв и ограниченная с каждой стороны пробелом, символом конца строки или другим

символом, не являющимся буквой. Программа должна повторять эти действия до тех пор, пока пользователь не захочет закончить работу.

6. Напишите функцию сортировки, похожую на функцию из листинга 10.10 главы 10, с той разницей, что ее аргументом должен быть вектор, содержащий числа типа `int`, а не массив. Этой функции не нужен параметр, подобный параметру `number_used` функции, приведенной в листинге 10.10, поскольку количество используемых элементов вектора всегда можно узнать с помощью его функции-члена `size()`. У вашей функции сортировки будет только один параметр, и он будет иметь тип вектора. Используйте алгоритм сортировки выбором, как в программе из листинга 10.10.
7. Еще раз реализуйте задание 5 из главы 10, но на этот раз вместо массивов используйте векторы. (Чтобы упростить задачу, сначала выполните предыдущее задание.)
8. Еще раз реализуйте задание 4 из главы 10, однако теперь вместо массивов воспользуйтесь векторами. Чтобы упростить задачу, сначала выполните задание 6 или 7 настоящей главы. Но учтите, что для этого задания вам придется писать собственный код сортировки, поскольку употреблявшаяся ранее функция сортировки в этом случае не подойдет. Можно использовать вектор со структурой в качестве базового типа.

Глава 12 Указатели и динамические массивы

Память необходима для любой деятельности разума.
Блез Паскаль

Указатель — это конструкция, которая обеспечивает дополнительные возможности управления памятью компьютера. В этой главе рассказывается, как указатели используются с массивами, и рассматривается новая форма массивов, называемых динамическими массивами. *Динамический массив* — это массив, размер которого определяется во время выполнения программы, а не задается при ее создании.

12.1. Указатели

Не указывайте пальцем в небо.
Дзен-буддистское изречение

Указатель — это адрес переменной в памяти. Напомним, что память компьютера разделена на нумерованные ячейки (именуемые байтами), а переменные реализуются как последовательности смежных ячеек. Если переменная занимает, скажем, три ячейки памяти, адрес первой из них иногда используется для доступа к переменной вместо имени. Например, когда переменная передается функции по ссылке, функция получает не имя переменной, а ее адрес.

Адрес переменной, используемый для доступа к ее значению, называется указателем, поскольку он «указывает» на переменную в памяти, то есть идентифицирует переменную, тем самым определяя ее местоположение. Так, чтобы сослаться на переменную с адресом 1007, можно сказать: «Вот та переменная, по адресу 1007».

Вы уже не раз пользовались указателями в разных ситуациях. В частности, когда переменная передается функции по ссылке, последняя получает указатель на эту

переменную. В таком случае указатель передается автоматически. Теперь вы узнаете, как можно манипулировать указателями по собственному усмотрению.

Переменные-указатели

Указатель может храниться в переменной. Но, хотя он и является адресом в памяти, а адрес представляет собой число, указатель не может храниться в переменной типа `int` или `double`. Для этого требуется переменная особого типа. Например, оператор

```
double *p;
```

объявляет переменную `p` как указатель на переменную типа `double`.

В этой переменной `p` могут храниться указатели на переменные типа `double`, но она не может содержать указатели на переменные других типов, таких как `int` или `char`. Каждому типу переменных соответствует свой тип указателей.

В общем случае объявление переменной для хранения указателей на другие переменные определенного типа выполняется точно так же, как объявление переменной базового типа, но перед ее именем добавляется символ звездочки. Так, следующий оператор:

```
int *p1, *p2, v1, v2;
```

объявляет переменные `p1` и `p2` как указатели на переменные типа `int`, а переменные `v1` и `v2` как обычные переменные типа `int`.

Перед каждой переменной-указателем должен стоять символ звездочки. Если из приведенного выше объявления удалить второй символ звездочки, `p2` будет обычной переменной типа `int`. Звездочка как обозначение указателя — это тот же символ, которым в языке C++ обозначается операция умножения, но в данном контексте его смысл совсем иной.

Объявление переменных-указателей

Переменная для хранения указателей на другие переменные типа *тип_данных* объявляется так же, как обычная переменная указанного типа данных, только перед ее именем добавляется символ звездочки.

Синтаксис

тип_данных `*`*имя_переменой_1*`, *`*имя_переменой_2*`, ...;`

Пример

```
double *pointer1, *pointer2;
```

При описании указателей и переменных для их хранения говорят, что переменная указывает на другую переменную, а не содержит ее адрес. Например, когда переменная-указатель `p1` содержит адрес переменной `v1`, говорят, что она *указывает* (ссылается) на переменную `v1`, или *является указателем* на переменную `v1`.

Адреса и числа

Указатель является адресом, а адрес представляется целым числом, но указатель не является целым числом. Это не парадокс — это абстракция! В C++ принято, чтобы указатели использовались исключительно в качестве адресов, а не в качестве чисел. Указатель не является значением типа int или какого-либо другого числового типа, и его нельзя хранить в переменной типа int. Если попытаться так сделать, большинство компиляторов выведет сообщение об ошибке или предупреждение. С указателями нельзя выполнять и обычные арифметические операции, а только особого рода сложение и вычитание, но это не то же самое, что целочисленное сложение и вычитание.

Переменные-указатели, например объявленные выше переменные p1 и p2, могут являться указателями на обычные переменные, такие, как v1 и v2. Адрес переменной для присваивания переменной-указателю можно узнать с помощью оператора &. Так, следующая строка кода:

```
p1 = &v1;
```

присваивает переменной p1 указатель на переменную v1.

Теперь на эту переменную можно ссылаться двумя способами: по имени или с помощью указателя p1 (то есть ее можно называть v1, а можно «переменной, на которую указывает p1»). В C++ обращение к переменной, на которую указывает p1, выполняется так: *p1. Здесь символ звездочки используют, как оператор *разыменования*, и тогда переменную-указатель именуют *разыменованной*.

С помощью описанных операций с указателями можно получать интересные результаты. Рассмотрим следующий код:

```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

Вот что он выводит на экран:

```
42
42
```

Если p1 содержит указатель на v1, то v1 и *p1 указывают на одну и ту же переменную. Поэтому, когда переменной *p1 присваивается значение 42, переменной v1 тоже присваивается данное значение.

Символ &, используемый для получения адреса переменной, — тот же символ, с помощью которого в объявлении функции определяются параметры, предаваемые по ссылке. И это не совпадение. Вспомните, что при передаче аргумента по ссылке в функцию передается его адрес. Поэтому эти два способа применения символа & очень близки, но все же не эквивалентны, и мы будем считать их различными.

Значение переменной-указателя можно присвоить иной переменной-указателю. В результате из одной переменной в другую будет скопирован хранящийся в ней адрес. Например, если p1 все еще указывает на v1, оператор

```
p2 = p1;
```

присвоит переменной p2 соответствующее значение, и она будет указывать на v1.

Если значение v1 еще не изменилось, то оператор

```
cout << *p2;
```

выведет на экран число 42.

С указателями нужно работать очень внимательно, нельзя путать операторы

```
p1 = p2;
```

и

```
*p1 = *p2;
```

Операторы * и &

Оператор *, стоящий перед переменной-указателем, возвращает переменную, на которую ссылается хранящийся в ней указатель. Когда оператор используется таким образом, его называют оператором *разыменования*.

Оператор &, введенный перед обычной переменной, возвращает ее адрес, то есть он возвращает указатель на переменную. Такой оператор именуют оператором *взятия адреса*.

В качестве примера рассмотрим следующее объявление:

```
double *p, v;
```

Оператор

```
p = &v;
```

присваивает переменной p такое значение, чтобы она указывала на переменную v.

Выражение *p возвращает переменную, на которую указывает переменная p, так что переменные *p и v содержат одно и то же значение. Например, оператор

```
*p = 9.99;
```

присваивает переменной v значение 99.9, хотя имя v в нем вообще не используется.

Когда перед именами переменных p1 и p2 стоят символы звездочки, мы имеем дело не с указателями, а с переменными, на которые они указывают (рис. 12.1).

Так как указатель может использоваться для доступа к переменной, программа имеет возможность работать даже с теми переменными, у которых нет собственных идентификаторов. Для создания переменной без идентификатора применяется оператор new. Операции с такими безымянными переменными выполняются только с помощью указателей. Следующий оператор:

```
p1 = new int;
```

создает новую переменную типа int и присваивает ее адрес переменной-указателю p1 (так что p1 указывает на эту новую безымянную переменную).

К созданной переменной можно обращаться таким образом: *p1 (то есть как к переменной, на которую указывает p1). С ней можно производить все действия, допустимые с другими переменными типа int. Скажем, вот такой код:

```
cin >> *p1;
*p1 = *p1 + 7;
cout << *p1;
```

помещает в нее значение типа int, введенное с клавиатуры, прибавляет к нему 7 и выводит результат.

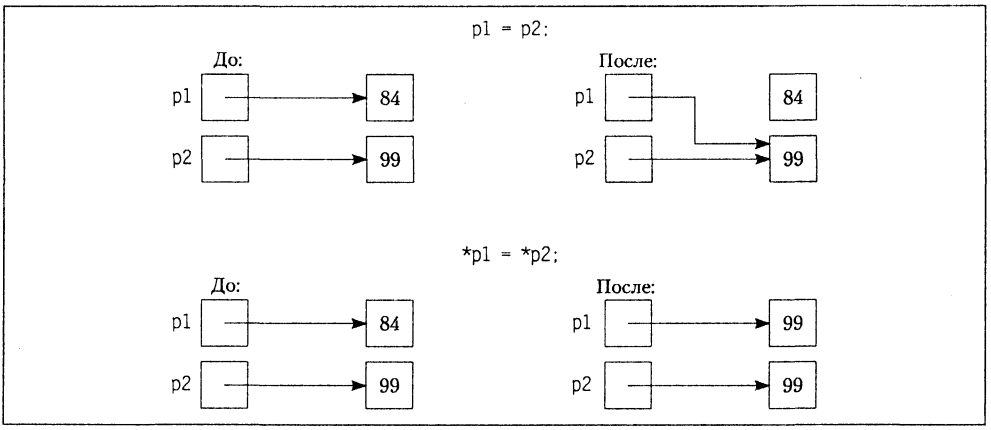


Рис. 12.1. Использование оператора присваивания

Оператор new создает новую безымянную переменную и возвращает указатель на нее. Тип этой переменной задается после ключевого слова new. Переменные, созданные с помощью оператора new, называются *динамическими переменными*, поскольку они создаются и удаляются во время выполнения программы. В листинге 12.1 приведена программа, демонстрирующая некоторые простейшие операции с указателями и динамическими переменными.

Листинг 12.1. Базовые операции с указателями

```
// Программа, демонстрирующая операции с указателями и динамическими переменными.
#include <iostream>
using namespace std;

int main()
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;
    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    p1 = new int;
    *p1 = 88;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    cout << "Hope you got the point of this example!\n";
    return 0;
}
```

Пример диалога

```
*p1 == 42
*p2 == 42
```

```
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

На рис. 12.2 показана схема операций, выполняемых программой из листинга 12.1. Переменные на этой схеме представлены квадратиками, внутри которых записаны соответствующие им значения. Мы не показываем реальные числовые адреса, хранящиеся в переменных-указателях, поскольку это не имеет значения. Важно только, что каждое из них является адресом конкретной переменной. Вместо реальных адресов на схеме нарисованы стрелки, соединяющие указатели с переменными, адреса которых они содержат. На рис. 12.2, *б* видно, что переменная `p1` содержит адрес переменой, в квадратике которой записан знак вопроса.

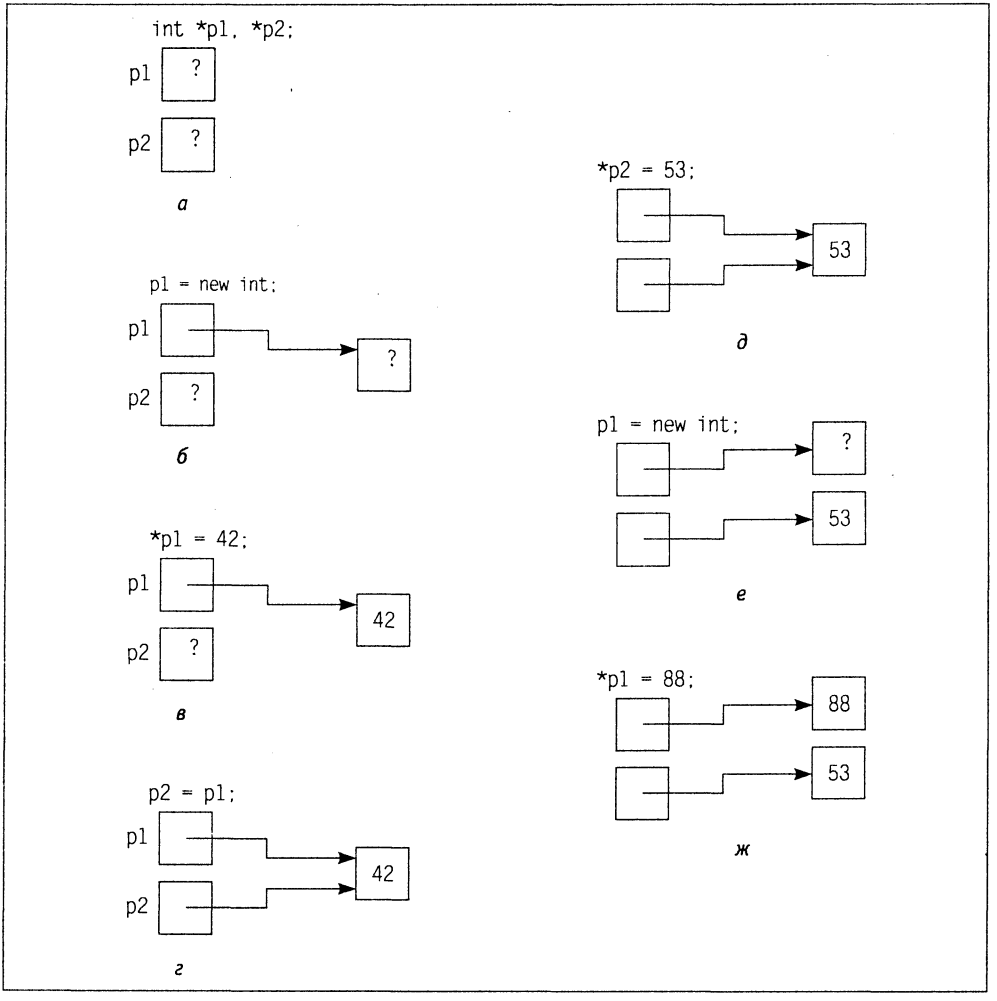


Рис. 12.2. Пояснение к программе из листинга 12.1.

Переменные-указатели и оператор =

Если p1 и p2 являются указателями, то оператор

```
p1 = p2;
```

изменяет значение переменной p1 так, чтобы она указывала на тот же объект, что и переменная p2.

Оператор new

Оператор new создает новую динамическую переменную заданного типа и возвращает указатель на нее. Например, следующий код:

```
MyType *p;
p = new MyType;
```

создает новую динамическую переменную типа MyType и присваивает указатель на нее переменой p.

Если типом переменной является класс с конструктором, то для ее создания вызывается используемый по умолчанию конструктор этого класса. Можно вызвать и другой конструктор, задав соответствующие инициализирующие значения:

```
int *n;
n = new int(17);              // Инициализирует переменную n значением 17.
MyType *mtPtr;
mtPtr = new MyType(32.0, 17); // Вызывает MyType(double, int);
```

Стандарт C++ определяет, что в случае если для создания новой переменной недостаточно памяти, оператор new по умолчанию должен завершить работу программы.

Упражнения для самопроверки

1. Расскажите о концепции использования указателей в C++.
2. Как можно ошибочно интерпретировать следующее объявление:

   ```
   int* int_ptr1, int_ptr2;
   ```

3. Укажите как минимум два способа применения символа *. Что делает каждый из этих операторов и как он называется?
4. Что выводит следующий код:

   ```
   int *p1, *p2;
   p1 = new int;
   p2 = new int;
   *p1 = 10;
   *p2 = 20;
   cout << *p1 << " " << *p2 << endl;
   p1 = p2;
   cout << *p1 << " " << *p2 << endl;
   *p1 = 30;
   cout << *p1 << " " << *p2 << endl;
   ```

   Как изменится его вывод, если заменить оператор

   ```
   *p1 = 30;
   ```

оператором

```
*p2 = 30;
```

5. Что выводит следующий код:

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
*p1 = *p2;   // Эта строка не такая, как в упражнении 4
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

Основы управления памятью

В C++ динамическим переменным предоставляется специальная область памяти, называемая *динамической областью* (от англ. freestore) или просто *кучей* (от англ. heap). Для каждой новой динамической переменной, которая создана программой, выделяется память из этой области. Если в программе будет создано слишком много динамических переменных, они займут всю память динамической области, после чего новые вызовы оператора new будут завершаться неудачей.

Размер динамической области зависит от конкретной реализации C++. Обычно он достаточно велик, и программе трудно использовать всю память этой области. Но даже в самых современных программах динамическую память, которая больше не нужна, следует возвращать системе. Чтобы освободить память, занимаемую динамической переменной, и возвратить ее в динамическую область для повторного выделения другим переменным, используют оператор delete. Предположим, что p — указатель на динамическую переменную. Тогда следующий оператор:

```
delete p;
```

удаляет эту переменную и возвращает занимаемую ею память в динамическую область.

После выполнения данного оператора значение переменной p становится не определенным и ее можно считать неинициализированной переменной.

Оператор delete

Оператор delete удаляет динамическую переменную и возвращает занимаемую ею память в динамическую область, после чего эту память можно использовать для создания новых динамических переменных. Например, следующий оператор:

```
delete p;
```

удаляет динамическую переменную, на которую указывает переменная p.

После выполнения данного оператора значение переменной p становится не определенным и ее можно считать неинициализированной переменной. (Далее в этой главе описывается несколько отличающаяся версия оператора delete, предназначенная для работы с динамической переменной-массивом.)

Ловушка: зависшие указатели

Когда оператор `delete` применяется к переменной-указателю, динамическая переменная, на которую она указывает, удаляется. В результате значение переменной-указателя становится не определенным, то есть неизвестно, ни куда она указывает, ни каково значение, на которое она указывает. Более того, если на удаленную динамическую переменную ссылается еще какой-нибудь указатель, его значение тоже не определено. Такие переменные-указатели с не определенными значениями называются *зависшими*. Если `p` – зависший указатель, и в программе к нему применяется оператор разыменования *, результат такого использования непредсказуем. Применяя оператор разыменования к переменной-указателю, нужно быть уверенным, что она указывает на существующую переменную.

Динамические и автоматические переменные

Переменные, создаваемые с помощью оператора `new`, называются динамическими, потому что они создаются и удаляются во время выполнения программы. В сравнении с ними обычные переменные можно было бы назвать статическими, но в терминологии C++ это название не используется.

Обычные переменные, с которыми мы работали в предыдущих главах, действительно не являются статическими. Если переменная локальна для функции, она создается C++ при вызове этой функции и уничтожается по завершении вызова. Поскольку главная часть программы представляет собой функцию `main`, то же самое касается переменных, объявленных в этой части программы. (Так как вызов функции `main` не завершается до тех пор, пока не завершится работа программы, объявленные в ней переменные тоже не удаляются до конца работы, но механизм их создания и удаления такой же, как в любой другой функции.) Обычные переменные, объявленные в функции `main` или любой другой функции программы, часто называют *автоматическими*, ведь они тоже имеют динамическую природу, но создаются и удаляются автоматически. Мы же будем называть их *обычными переменными*, хотя в других книгах часто применяют термин «автоматические».

Существует еще одна категория переменных – *глобальные переменные* программы. Такие переменные объявляются вне определений всех функций, в том числе вне функции `main` (мы кратко рассказывали о них в главе 3.) В примерах этой книги потребность в глобальных переменных не возникает, поэтому мы их не используем.

Совет программисту: определяйте типы указателей

Язык C++ позволяет определить тип указателей и использовать его в объявлениях переменных для того, чтобы перед именами таких переменных не приходилось помещать символ звездочки. Например, оператор

```
typedef int* IntPtr;
```

определяет тип данных `IntPtr` для указателей на переменные типа `int`.

После обработки этой строки кода следующие два объявления переменных-указателей:

```
IntPtr p;
```

и

```
int *p;
```

будут эквивалентны.

С помощью оператора typedef можно определить псевдоним любого имени или определения типа. Так, оператор

```
typedef double Kilometers;
```

определяет имя типа Kilometers, означающее то же самое, что и double.

После этого определения можно объявлять переменные типа double следующим образом:

```
Kilometers distance;
```

Возможность переименовывать существующие типы данных иногда может быть полезна, но важнейшим применением ключевого слова typedef является определение типов для переменных-указателей.

У типов указателей, определенных программистом (таких как приведенный выше тип IntPtr), имеется два преимущества. Во-первых, они позволяют избежать маленькой, но вредоносной ошибки – случайно забытого символа звездочки. Скажем, если программист собирался объявить переменные p1 и p2 как указатели, следующая строка:

```
int *p1, p2;
```

будет ошибкой.

Поскольку перед именем переменной p2 нет символа звездочки, она объявляется как обычная переменная типа int, а совсем не как указатель. Если же символ звездочки сдвинуть влево, как в следующем примере, результат будет таким же, но заметить ошибку будет еще сложнее. C++ позволяет расположить относящуюся к переменной звездочку после ключевого слова int без использования пробела. Так что оператор

```
int* p1, p2;
```

допустим, но его синтаксис сбивает с толку. Может показаться, что переменные p1 и p2 объявляются как указатели, а на самом деле p1 объявляется как указатель, а p2 – как обычная переменная типа int. С точки зрения компилятора символ звездочки вовсе не относится к ключевому слову int, хотя и стоит рядом с ним, а относится к идентификатору p1. Вот как правильно объявить указатели p1 и p2:

```
int *p1, *p2;
```

Более простой и надежный с точки зрения риска случайных ошибок описанного рода способ объявления указателей p1 и p2 заключается в использовании определенного программистом типа IntPtr:

```
IntPtr p1, p2;
```

Второе преимущество определенного программистом типа указателей, подобного IntPtr, становится очевидным, когда определяется функция с параметром-указателем, передаваемым по ссылке. При отсутствии именованного типа указателей в определение функции (точнее, ее параметра) приходится включать и символ звездочки, и символ амперсанда, что ухудшает его читабельность. Если же воспользоваться именованным типом, определение передаваемого по ссылке параметра будет выглядеть как обычно. Например:

```
void sample_function(IntPtr& pointer_variable);
```

Определение типа

Любому из существующих типов данных можно присвоить новое имя и использовать его для объявления переменных. Такое имя объявляется с помощью ключевого слова typedef. Подобные определения типов обычно помещаются вне тела функции main (и конечно, вне других функций) — там же, где определяются структуры и классы. Мы будем использовать определяемые программистом имена типов для указателей так, как показано в приведенном ниже примере.

Синтаксис

```
typedef тип_данных новое_имя_типа;
```

Пример

```
typedef int* IntPtr;
```

Теперь имя типа IntPtr можно применять для объявления указателей на динамические переменные типа int, вот так:

```
IntPtr pointer1, pointer2;
```

Упражнения для самопроверки

6. Предположим, что динамическая переменная в программе создана следующим образом:

   ```
   char *p;
   p = new char;
   ```

   Если значение указателя p не изменилось, то есть он ссылается на ту же самую динамическую переменную, каким образом ее можно удалить и вернуть занимаемую ею память в динамическую область, чтобы эта память могла использоваться для создания новых динамических переменных?

7. Напишите определение типа NumberPtr для переменных, назначение которых — хранить указатели на динамические переменные типа int. Затем напишите объявление переменной-указателя my_point типа NumberPtr.

8. Расскажите о действии оператора new. Что он возвращает?

12.2. Динамические массивы

В этом разделе вы узнаете, что переменные-массивы на самом деле являются указателями. Кроме того, мы рассмотрим, как пишутся программы с динамическими

массивами. *Динамический массив* — это массив, размер которого определяется не при написании программы, а при ее выполнении.

Массивы переменных и переменные-указатели

В главе 10 говорилось, как массивы хранятся в памяти. Тогда вы еще не имели представления об указателях, и поэтому мы рассказывали о массивах, используя терминологию адресов в памяти. Но из материала этой главы вы уже знаете, что адрес в памяти является указателем. Поэтому в C++ массив переменных на самом деле является указателем, ссылающимся на первый элемент массива. Так следующие два объявления:

```
int a[10];
typedef int* IntPtr;
IntPtr p;
```

создают переменные одного и того же типа.

То, что a и p — переменные одного типа, подтверждает программа, приведенная в листинге 12.2. Поскольку a — указатель на переменную типа int (на массив a[0]), его значение можно присвоить указателю p, как в приведенном ниже примере:

```
p = a;
```

Листинг 12.2. Массивы как переменные-указатели

```
// Программа, показывающая, что переменная типа массива является указателем.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main()
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;
    p = a;

    for (index = 0; index < 10; index++)
        cout << p[index] << " ";
    cout << endl;

// Обратите внимание, что изменения в массиве p одновременно
// являются и изменениями в массиве a.
    for (index = 0; index < 10; index++)
        p[index] = p[index] + 1;

    for (index = 0; index < 10; index++)
        cout << a[index] << " ";
    cout << endl;

    return 0;
}
```

Вывод

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

После присваивания переменная p указывает на ту же ячейку памяти, что и a, поэтому выражения p[0], p[1], ..., p[9] являются ссылками на элементы массива a[0], a[1], ..., a[9]. Этот синтаксис с использованием квадратных скобок, который вы привыкли применять, работая с массивами, годится и при работе с указателями, ссылающимися на массивы. После выполнения приведенного выше присваивания можно считать p идентификатором массива. В то же время идентификатор a можно рассматривать как указатель, хотя здесь имеется важное ограничение: значение указателя в переменной, идентифицирующей массив (такой, как a), нельзя изменить. Вы можете предположить, что допустимо следующее присваивание:

```
IntPtr p2;
// Переменной p2 присваивается некоторое значение указателя.
a = p2; // НЕДОПУСТИМО (Переменной a нельзя присвоить другой адрес.)
```

но это не так, на что указывает комментарий.

Создание и использование динамических массивов

Одним из недостатков массивов, с которыми мы работали до сих пор, является следующее: их размер задается раз и навсегда при написании программы, тогда как во многих случаях требуемый размер массива становится известен только во время выполнения программы. Допустим, массив предназначен для хранения списка порядковых номеров студентов, но запуск программы может каждый раз относиться к другому классу со своим количеством студентов. И если использовать уже знакомый нам тип массивов, придется оценить максимально возможное количество студентов в классе и объявить массив максимального размера, надеясь, что программа никогда не будет запущена для еще большего класса. Очевидно, что здесь возникают две проблемы. Во-первых, можно ошибиться в оценке максимального размера массива, и тогда в некоторых случаях программа не будет работать. Во-вторых, поскольку в массиве может быть много неприменяемых элементов, программа будет нерационально использовать память компьютера. Обе эти проблемы решаются с помощью динамических массивов. Если для списка студентов в программе создается такой массив, количество учащихся в классе можно определять на входе программы, а затем создавать массив именно такого размера.

Динамические массивы создаются с помощью оператора new, и работать с ними достаточно легко. Поскольку массивы переменных являются указателями, с помощью названного оператора можно создавать массивы динамических переменных и обращаться с ними, как с обычными массивами. Например, следующий код:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

создает массив динамических переменных с десятью элементами типа double.

Для получения динамического массива элементов любого другого типа достаточно просто заменить double именем этого типа. Так, тип double можно заменить типом структуры или класса. Для получения динамического массива другого размера нужно указать этот размер вместо значения 10.

Как пользоваться динамическими массивами

- **Определить тип указателя.** Определить тип для указателей на переменные того типа, к которому относятся элементы массива. В частности, если динамический массив должен быть массивом типа double, можно использовать оператор

  ```
  typedef double* DoubleArrayPtr;
  ```

- **Объявить переменную-указатель.** Объявить переменную-указатель определенного выше типа. Эта переменная будет ссылаться на динамический массив в памяти и служить именем этого массива:

  ```
  DoubleArrayPtr a;
  ```

- **Вызвать оператор new.** Создать динамический массив с помощью оператора new:

  ```
  a = new double[array_size];
  ```

  Размер динамического массива задается в квадратных скобках, как в приведенном примере. Его можно определить с помощью переменной типа int или выражения типа int. В данном примере array_size является переменной типа int, значение которой определяется во время выполнения программы.

- **Использовать как обычный массив.** Переменная-указатель используется подобно обычному массиву. Так, обращение к элементам массива производится с помощью обычного синтаксиса: a[0], a[1] и т. д. Переменной-указателю не может быть присвоено никакое другое значение, но она может применяться как обычный массив переменных.

- **Вызвать delete [].** Когда программа завершит работу с динамической переменной, следует вызвать оператор delete с пустыми квадратными скобками и переменной-указателем, чтобы удалить динамический массив и вернуть занимаемую им память в динамическую область. Например:

  ```
  delete [] a;
  ```

Этот пример имеет несколько не очевидных особенностей. Во-первых, тип данных, используемый для создания указателя на динамический массив, ничем не отличается от типа, который можно было бы применить для указателя на один элемент массива. Скажем, тип указателя на массив элементов типа double можно использовать и для создания указателей на обычные переменные этого типа. Указатель на массив — это на самом деле указатель на первый элемент массива. В приведенном выше примере создается массив с десятью элементами, но указатель p попрежнему ссылается на первый из них.

Обратите также внимание на то, что при вызове оператора new размер динамического массива задается в квадратных скобках за именем типа, в данном случае после ключевого слова double. Он говорит компилятору, сколько памяти нужно выделить для динамического массива. Если удалить из оператора квадратные скобки и число 10, он выделит память только для одной переменной типа double, а не для

массива из десяти элементов этого типа. Как вы увидите в программе из листинга 12.3, вместо константы 10 можно использовать переменную типа int, благодаря чему размер массива будет определяться во время выполнения программы, например путем ввода. Данная программа запрашивает у пользователя количество чисел в массиве, и затем с помощью оператора new создает динамический массив определенного размера. Размер этого массива задается в виде значения переменной array_size.

Листинг 12.3. Динамический массив

```
// Сортирует список чисел, введенных пользователем с клавиатуры.
#include <iostream>
#include <cstdlib>
#include <cstddef>

typedef int* IntArrayPtr;

void fill_array(int a[], int size);          // Обычные параметры типа массива.
// Предусловие: size - это размер массива a.
// Постусловие: элементам от a[0] до a[size-1]
// присвоены значения, введенные с клавиатуры.

void sort(int a[], int size);                // Обычные параметры типа массива.
// Предусловие: size - это размер массива a.
// Элементам от a[0] до a[size-1] присвоены значения.
// Постусловие: значения элементов от a[0] до a[size-1]
// реорганизованы так, что a[0] <= a[1] <= ... <= a[size-1].

int main()
{
    using namespace std;
    cout << "This program sorts numbers from lowest to highest.\n";

    int array_size;
    cout << "How many numbers will be sorted? ";
    cin >> array_size;

    IntArrayPtr a;
    a = new int[array_size];

    fill_array(a, array_size);
    sort(a, array_size);

    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < array_size; index++)
        cout << a[index] << " ";  // Динамический массив a применяется так же, как обычный.
    cout << endl;

    delete [] a;

    return 0;
}

// Используем библиотеку классов iostream.
void fill_array(int a[], int size)
```

```
{
    using namespace std;
    cout << "Enter " << size << " integers.\n";
    for (int index = 0; index < size; index++)
        cin >> a[index];
}

void sort(int a[], int size)
```

... Может использоваться любая реализация функции sort. Она может требовать или не требовать определения дополнительных функций. И какой бы ни была эта реализация, она не нуждается в сведениях о том, что в первом аргументе ей передается не обычный, а динамический массив. Попробуйте применить реализацию, приведенную в листинге 10.10 (с подходящими именами параметров). ...

Обратите внимание на оператор delete, удаляющий динамический массив a в программе из листинга 12.3. В нем нет необходимости, поскольку выполнение программы и так уже завершается. Но если бы программа продолжала работу и выполняла другие операции с динамическими переменными, этот оператор был бы полезен для удаления динамического массива и освобождения занимаемой им памяти. Оператор delete для динамического массива подобен уже знакомому вам оператору delete для обычных переменных и отличается только наличием пустых квадратных скобок:

```
delete [] a;
```

Эти скобки указывают компилятору C++, что удаляется динамический массив, и система должна выяснить размер данного массива и удалить все его элементы. Если опустить квадратные скобки, компьютер получит указание удалить только одну переменную типа int. Например, использование оператора

```
delete a;
```

недопустимо, но большинство компиляторов не обнаружит ошибки. В стандарте ANSI C++ говорится, что последствия такого вызова не определены, это означает, что создатель компилятора может поступить, как сочтет удобным (для себя, а не для программистов). Если компилятор в этом случае и произведет какие-то полезные действия, нет никакой гарантии, что его следующая версия или другой компилятор сделают то же самое. Вывод прост: для освобождения памяти, выделенной динамическому массиву с помощью оператора, подобного приведенному ниже:

```
array_ptr = new MyType[37];
```

всегда используйте такой синтаксис:

```
delete [] array_ptr;
```

Динамический массив создается посредством оператора new с использованием указателя, такого, как a в листинге 12.3. После вызова названного оператора не следует присваивать переменной-указателю никакое другое значение, поскольку иначе при выполнении оператора delete система не сможет правильно освободить занимаемую массивом память.

В остальном динамический массив используется так же, как обычный.

Упражнения для самопроверки

9. Напишите определение типа для переменных-указателей на динамический массив, типом элементов которого является char. Назовите этот тип CharArray.

10. Предположим, что программа содержит следующий код, создающий динамический массив:

```
int *entry;
entry = new int[10];
```

В результате его выполнения создается переменная entry, указывающая на динамический массив. Напишите код для заполнения этого массива десятью числами, введенными с клавиатуры.

11. Предположим, что программа содержит код для создания динамического массива, как в упражнении 10, и значение переменной-указателя entry еще не изменилось. Напишите код для удаления этого массива и возвращения занимаемой им памяти в динамическую область.

12. Что выводит следующий фрагмент кода:

```
int a[10];
int *p = a;
int i;
for (i = 0; i < 10; i++)
    a[i] = i;
for (i = 0; i < 10; i++)
    cout << p[i] << " ";
cout << endl;
```

Предполагается, что он выполняется в составе полной программы.

13. Что выводит следующий фрагмент кода:

```
int array_size = 10;
int *a;
a = new int[array_size];
int *p = a;
int i;
for (i = 0; i < array_size; i++)
    a[i] = i;
p[0] = 10;
for (i = 0; i < array_size; i++)
    cout << a[i] << " ";
cout << endl;
```

Предполагается, что код выполняется в составе полной программы.

Арифметические операции с указателями (факультативный материал)

Существует ряд арифметических операций, которые можно выполнять с указателями, но это операции с адресами, а не с числами. В качестве примера предположим, что программа содержит следующий код:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

После его выполнения в переменной d будет содержаться адрес элемента d[0]. Выражение d+1 возвращает адрес элемента d[1], выражение d+2 — адрес элемента d[2] и т. д. Заметьте, что хотя значением переменной d является адрес, и при этом он представляет собой число, операция d+1 не прибавляет единицу к числу, хранящемуся в переменной d. Если для переменной типа double требуется восемь байтов (восемь ячеек памяти) и d содержит адрес 2001, то выражение d+1 возвращает адрес 2009. Конечно, вместо типа double может использоваться другой тип данных, и тогда значение указателя будет изменяться в соответствии с размером, требуемым для значений этого типа.

Описанные арифметические операции с указателями предоставляют альтернативный способ доступа к элементам массивов. Так, если array_size — размер динамического массива, на который ссылается указатель d, следующий цикл:

```
for (int i = 0; i < array_size; i++)
    cout << *(d + i)<< " ";
```

выведет содержимое этого массива на экран.

Приведенный цикл эквивалентен такому:

```
for (int i = 0; i < array_size; i++)
    cout << d[i] << " ";
```

Операции умножения и деления с указателями не выполняются. Допускается только прибавление к указателю целого числа, вычитание из него целого числа и вычисление разности для двух указателей одного типа. Разность равна количеству элементов массива между двумя адресами. Помните, что указатели, для которых определяется разность, должны ссылаться на один и тот же массив, поскольку значение разности, полученное для указателей на разные массивы, не имеет смысла. С указателями можно использовать операторы инкрементирования (++) и декрементирования (--). Так, выражение d++ увеличивает значение указателя d, помещая в него адрес следующего элемента массива, а выражение d-- помещает в указатель d адрес предыдущего элемента массива.

Упражнения для самопроверки

Эти упражнения относятся к факультативному разделу, посвященному арифметическим операциям с указателями.

14. Что выводит следующий фрагмент кода:

```
int array_size = 10;
int *a;
a = new int[array_size];
int i;
for (i = 0; i < array_size; i++)
    *(a + i) = i;

for (i = 0; i < array_size; i++)
    cout << a[i] << " ";
cout << endl;
```

выполненный в составе полной программы?

15. Что выводит следующий фрагмент кода:

```
int array_size = 10;
int *a;
a = new int[array_size];
int i;
for (i = 0; i < array_size; i++)
    a[i] = i;

while (*a < 9)
{
    a++;
    cout << *a << " ";
}
cout << endl;
```

выполненный в составе полной программы?

Многомерные динамические массивы (факультативный материал)

C++ позволяет определять многомерные динамические массивы. Напомним, что многомерный массив — это массив массивов, или массив массивов массивов и т. д. Например, для создания двумерного динамического массива, учитывая, что это массив массивов, сначала нужно создать одномерный динамический массив указателей типа `int*` (то есть типом динамического одномерного массива должен являться `int`), затем для каждого элемента данного массива указателей создать одномерный динамический массив значений типа `int`.

Эту задачу может облегчить определение типа для обычных динамических массивов типа `int`:

```
typedef int* IntArrayPtr;
```

Если в программе есть такое определение, для получения массива значений типа `int` размером 3×4 необходимо сначала создать массив базового типа `IntArrayPtr`:

```
IntArrayPtr *m = new IntArrayPtr[3];
```

Это массив из трех указателей на динамические массивы типа `int`, и его следует заполнить массивами из четырех элементов следующим образом:

```
for (int i = 0; i < 3; i++)
    m[i] = new int[4];
```

Результирующий массив `m` является динамическим массивом размером 3×4. Его создание и использование демонстрирует программа, приведенная в листинге 12.4.

Обратите внимание на использование в ней оператора `delete`. Так как динамический массив является массивом массивов, каждый вложенный массив, созданный с помощью оператора `new` в цикле `for`, нужно удалить, используя отдельный вызов оператора `delete[]`, после чего посредством еще одного вызова названного оператора уничтожить внешний массив. В результате вся занимаемая им память возвращается в динамическую область. (Поскольку программа завершается сразу после вызовов `delete[]`, их можно опустить, но наша задача — продемонстрировать, как они обрабатываются.)

Листинг 12.4. Двумерный динамический массив

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

typedef int* IntArrayPtr;

int main()
{
    int d1, d2;
    cout << "Enter the row and column dimensions of the array:\n";
    cin >> d1 >> d2;

    IntArrayPtr m = new IntArrayPtr[d1];
    int i, j;
    for (i = 0; i < d1; i++)
        m[i] = new int[d2];
    // Теперь m является массивом d1 на d2.

    cout << "Enter " << d1 << " rows of "
         << d2 << " integers each:\n";
    for (i = 0; i < d1; i++)
        for (j = 0; j < d2; j++)
            cin >> m[i][j];

    cout << "Echoing the two-dimensional array:\n";
    for (i = 0; i < d1; i++)
    {
        for (j = 0; j < d2; j++)
            cout << m[i][j] << " ";
        cout << endl;
    }

/* Обратите внимание, что каждому вызову оператора new, создающему новый массив,
   соответствует отдельный вызов оператора delete[]. (Поскольку выполнение программы
   на этом завершается, вызовы delete[] в ней не требуются, но в другом контексте
   они были бы необходимы.) */

    for (i = 0; i < d1; i++)
        delete[] m[i];
    delete[] m;

return 0;
}
```

Пример диалога

```
Enter the row and column dimensions of the array:
3 4
Enter 3 rows of 4 integers each:
1 2 3 4
5 6 7 8
9 0 1 2
Echoing the two-dimensional array:
1 2 3 4
5 6 7 8
9 0 1 2
```

12.3. Классы и динамические массивы

Когда есть все возможности и средства.
Уильям Шекспир

Базовым типом динамического массива может быть класс, в то же время и класс может содержать динамический массив в качестве переменной-члена. К вашим услугам все технологии классов и динамических массивов в любом мыслимом сочетании. Давайте начнем с примера.

Пример: класс для строковой переменной

Из главы 11 мы узнали, каким образом определяются переменные массивы для хранения строк С, а в предыдущем разделе рассказывалось о создании динамических массивов, размер которых задается во время выполнения программы. В этом примере мы определим класс `StringVar`, объекты которого являются строковыми переменными. Объект этого класса будет реализован с использованием динамического массива, размер которого определяется во время выполнения программы. Поэтому объекты типа `StringVar` будут обладать всеми достоинствами динамических массивов плюс некоторыми дополнительными возможностями. Мы определим для класса `StringVar` функции-члены для работы с данными его объектов, выводящие сообщение об ошибке в ответ на попытку присвоить объекту слишком длинную строку. Конечно, так будет реализовано лишь небольшое подмножество всех возможных операций со строковыми объектами. В практическом задании 1 вам будет предложено расширить определение этого класса путем добавления новых функций-членов и перегруженных операторов.

Поскольку в C++ имеется стандартный класс `string`, описанный в главе 11, нет необходимости создавать класс `StringVar`. Но мы все-таки создадим его, это поможет лучше освоить основные принципы работы с классами и динамическими массивами.

Интерфейс типа `StringVar` приведен в листинге 12.5. Один из его конструкторов принимает единственный аргумент типа `int`. Данный аргумент определяет максимально допустимую длину строкового значения, которое может храниться в объекте. Конструктор, используемый по умолчанию, создает объект максимальной длины в 100 символов. Еще один конструктор принимает аргумент-массив, содержащий строку С (типа, описанного в главе 11). Заметьте, что аргументом такого конструктора может быть заключенная в кавычки литеральная строка. Этот конструктор инициализирует объект таким образом, что в нем может содержаться любая строка, размер которой не больше значения аргумента, и копирует в объект это значение аргумента. На конструкторе, названном конструктором копирования, мы пока останавливаться не будем. Не рассматривается и функция-член `~StringVar`, которая хотя с виду и похожа на конструктор, на самом деле таковым

не является. К ней мы вернемся несколько позже. Назначение остальных функций-членов класса StringVar очевидно.

В листинге 12.6 приведена простая программа, демонстрирующая использование класса StringVar. Два объекта, your_name и our_name, объявлены внутри определения функции conversation. В первом из них может содержаться любая строка, размер которой не превышает значение переменной max_name_size. Объект our_name инициализируется строковым значением "Borg", и его значение может быть заменено любым другим строковым значением длиной не более 4 символов.

Как указано в начале этого раздела, в основе класса StringVar лежит динамический массив. Реализация класса приведена в листинге 12.7. При объявлении объекта типа StringVar вызывается конструктор для его инициализации. Затем с помощью оператора new конструктор создает динамический массив символов для переменной-члена value. Строковое значение хранится в массиве value как обычная строка. Обратите внимание, что размер этого массива определяется только при объявлении объекта, когда вызывается конструктор и ему передается соответствующий аргумент. Как показывает программа из листинга 12.6, аргументом конструктора может быть переменная типа int. Посмотрите на объявление объекта your_name в определении функции conversation. В качестве аргумента конструктору передается полученный по значению параметр max_name_size. Напомним, что параметр, полученный по значению, является локальной переменной, поэтому max_name_size — это переменная. Таким образом, в качестве аргумента конструктора может использоваться переменная типа int.

Реализация функций-членов length, input_line и перегруженного оператора вывода << проста и не требует описания. В следующих разделах рассматривается функция ~StringVar и конструктор копирования.

Листинг 12.5. Файл интерфейса для класса StringVar

```
// Заголовочный файл strvar.h содержит интерфейс класса StringVar,
// объекты которого представляют строки.
// Обратите внимание, что максимальный размер
// строки задается как (max_size), а не как [max_size]:
//     StringVar the_object(max_size);
// где max_size - максимально допустимая длина строки.
#ifndef STRVAR_H
#define STRVAR_H
#include <iostream>
using namespace std;
namespace strvarsavitch
{

    class StringVar
    {
    public:
        StringVar(int size);
        // Инициализирует объект таким образом, чтобы он мог
        // вмещать строки, длина которых меньше значения аргумента size символов.
        // Устанавливает значение объекта равным пустой строке.
```

продолжение

Листинг 12.5 (*продолжение*)

```
        StringVar();
        // Инициализирует объект таким образом, чтобы он мог
        // вмещать строки длиной до 100 символов.
        // Устанавливает значение объекта равным пустой строке.

        StringVar(const char a[]);
        // Предусловие: массив a содержит строку символов,
        // завершающуюся символом '\0'.
        // Инициализирует объект таким образом, чтобы его значением
        // стала хранящаяся в массиве a строка и впоследствии ему можно было
        // присваивать строковые значения длиной до strlen(a) символов.

        StringVar(const StringVar& string_object);
        // Конструктор копирования.

        ~StringVar();
        // Возвращает используемую объектом память в динамическую область.

        int length() const;
         // Возвращает текущую длину строкового значения.

         void input_line(istream& ins);
        // Предусловие: если ins – входной файловый поток,
        // он уже соединен с файлом.
        // Действие: текст во входном потоке ins до символа '\n'
        // копируется в вызывающий объект. Если в нем недостаточно места,
        // копируется столько символов, сколько помещается в объект.

        friend ostream& operator <<(ostream& outs, const StringVar& the_string);
        // Перегружает оператор <<, чтобы им можно было пользоваться
        // для ввода значений типа StringVar.
        // Предусловие: если outs – выходной файловый поток,
        // он уже соединен с файлом.

    private:
        char *value;       // Указатель на динамический массив,
                           // содержащий строковое значение.
        int max_length;    // Максимально допустимая
                           // длина строкового значения.
    };
}   // strvarsavitch

#endif // STRVAR_H
```

Листинг 12.6. Программа, использущая класс StringVar

```
// Прикладная программа, демонстрирующая использование класса StringVar.
#include <iostream>
#include "strvar.h"

void conversation(int max_name_size);
// Осуществляет диалог с пользователем.

int main()
```

```
{
    using namespace std;
    conversation(30); // По завершении вызова функции память возвращается
                      // в динамическую область.
    cout << "End of demonstration.\n";
    return 0;
}

// Это только демонстрационная функция.
void conversation(int max_name_size)
{
    using namespace std;
    using namespace strvarsavitch;

    StringVar your_name(max_name_size), our_name("Borg"); // Определяет размер динамического
                                                          // массива.
    cout << "What is your name?\n";
    your_name.input_line(cin);
    cout << "We are " << our_name << endl;
    cout << "We will meet again " << your_name << endl;
}
```

Пример диалога

```
What is your name?
Kathryn Janeway
We are Borg
We will meet again Kathryn Janeway
End of demonstration
```

Листинг 12.7. Реализация класса StringVar

```
// Это файл реализации: strvar.cpp.
// (В вашей системе может потребоваться расширение, отличное от .cpp.)
// Это реализация класса StringVar.
// Интерфейс класса StringVar находится в заголовочном файле strvar.h.
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include <cstring>
#include "strvar.h"
using namespace std;

namespace strvarsavitch
{
    // Используем библиотеки классов cstddef и cstdlib.
    StringVar::StringVar(int size) : max_length(size)
    {
        value = new char[max_length + 1];//+1 для '\0'.
        value[0] = '\0';
    }

    // Используем библиотеки классов cstddef и cstdlib.
    StringVar::StringVar() : max_length(100)
    {
        value = new char[max_length + 1];//+1 для '\0'.
        value[0] = '\0';
    }
```

продолжение

Листинг 12.7 (*продолжение*)

```
    // Используем библиотеки классов cstring, cstddef и  cstdlib.
    StringVar::StringVar(const char a[]) : max_length(strlen(a))
    {
        value = new char[max_length + 1];   // +1 для '\0'.
        strcpy(value, a);
    }

    // Используем библиотеки классов cstring, cstddef и cstdlib.
    StringVar::StringVar(const StringVar& string_object)
                          : max_length(string_object.length())
    {
        value = new char[max_length + 1]; // +1 для '\0'.
        strcpy(value, string_object.value);
    }

     StringVar::~StringVar()
    {
        delete [] value;   // Деструктор
    }

    // Используем библиотеку классов cstring.
    int StringVar::length() const
    {
        return strlen(value);
    }

    // Используем библиотеку классов iostream.
    void StringVar::input_line(istream& ins)
    {
        ins.getline(value, max_length + 1);
    }

    // Используем библиотеку классов iostream.
    ostream& operator <<(ostream& outs, const StringVar& the_string)
    {
        outs << the_string.value;
        return outs;
    }

}   // strvarsavitch
```

Деструкторы

При использовании динамических переменных возникает одна существенная проблема. Их нельзя удалить, пока в программе не будет явно выполнен оператор `delete`. Даже если такая переменная создана с применением локальной переменной-указателя (которая, как положено всякой локальной переменной, существует до завершения вызова функции), она не удаляется, пока не будет вызван оператор `delete`. Без этого вызова переменная будет занимать память, и когда таких переменных накопится слишком много, они могут заполнить всю динамическую область и вызвать аварийное завершение работы программы. Более того, если динамическая переменная является членом класса, программист, который использует этот класс, даже не знает о ее существовании и, соответственно, не может вызвать для нее оператор `delete`. Поскольку переменные-члены класса обычно объявляют

закрытыми, программист не имеет доступа к необходимым указателям и не может вызвать для них оператор delete. Для решения этих проблем в C++ предусмотрен специальный вид функций-членов, называемых *деструкторами*.

Деструктор — это функция-член класса, автоматически вызываемая при выходе объекта данного класса из области видимости программы. Это означает, что если в некоторой функции программы имеется локальная переменная, то есть в данном случае объект с деструктором, по завершении вызова функции автоматически будет вызван деструктор объекта. И если такой деструктор определен верно, он с помощью операторов delete освобождает память, занимаемую всеми динамическими переменными-членами объекта. В зависимости от количества этих переменных может потребоваться один или несколько вызовов оператора delete. Деструктор выполняет и другие «уборочные» действия, но главная его задача — возвращение памяти в динамическую область.

Функция-член ~StringVar является деструктором класса StringVar, интерфейс которого приведен в листинге 12.5. Подобно конструктору, деструктор всегда имеет то же имя, что и класс, но с префиксом ~ (символ тильды), отличающим его от конструктора. Как и конструктор, он не имеет возвращаемого типа данных, и его тип вообще не указывается, даже как void. У деструктора отсутствуют параметры, поэтому класс может иметь только один деструктор, который нельзя перегружать. Если классу необходим еще один деструктор, он может быть определен как обычная функция-член.

Деструктор

Деструктор — это функция-член класса, автоматически вызываемая при выходе объекта данного класса из области видимости программы. Помимо прочего это означает, что если объект является локальной переменной функции, деструктор автоматически вызывается перед завершением ее вызова. Деструкторы предназначены для удаления созданных объектами динамических переменных и возврата занимаемой ими памяти в динамическую область. Они могут выполнять и другие действия, необходимые перед удалением объекта. Имя деструктора должно включать символ тильды (~), за которым следует имя класса.

Обратите внимание на определение деструктора ~StringVar, приведенное в листинге 12.7, — он вызывает оператор delete для удаления динамического массива, на который указывает переменная-член value. Вернитесь к определению функции conversation в программе из листинга 12.6 — ее локальные переменные your_name и our_name используются для создания динамических массивов. Если бы у класса StringVar не было деструктора, то после вызова функции conversation эти два динамических массива по-прежнему занимали бы память, будучи для программы бесполезными. Наша демонстрационная программа после вызова функции conversation почти сразу завершается, поэтому пара оставшихся в памяти массивов не представляет для нее угрозы. Но если класс StringVar без деструктора будет использоваться в реальной программе с большим количеством вызовов подобных функций, они исчерпают всю память динамической области, а это приведет к аварийному завершению работы программы.

Ловушка: указатели как параметры, передаваемые по значению

Если указатель передается в функцию по значению, это может вызвать проблемы. Рассмотрим вызов функции sneaky в программе из листинга 12.8. Параметр temp указанной функции передается по значению и поэтому является локальной переменной. При вызове функции данному параметру присваивается значение аргумента p, после чего выполняется тело функции. Никакие изменения значения параметра temp, естественно, не выходят за пределы функции sneaky. Таким образом, функция sneaky не может изменить значение переменной-указателя p. Однако из примера диалога можно сделать вывод, что значение этого указателя изменилось. Перед вызовом функции sneaky значение *p было равно 77, а после ее вызова стало равным 99. Что же произошло?

Листинг 12.8. Параметр-указатель, передаваемый по значению

```
// Программа, демонстрирующая поведение параметров-указателей, передаваемых по значению.
#include <iostream>
using namespace std;

typedef int* IntPointer;

void sneaky(IntPointer temp);

int main()
{
    IntPointer p;

    p = new int;
    *p = 77;
    cout << "Before call to function *p == "
         << *p << endl;

    sneaky(p);

    cout << "After call to function *p == "
         << *p << endl;

    return 0;
}

void sneaky(IntPointer temp)
{
    *temp = 99;
    cout << "Inside function call *temp == "
         << *temp << endl;
}
```

Пример диалога

```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

Действия программы, которая приведена в листинге 12.8, проиллюстрированы на рис. 12.3. Хотя вывод программы выглядит так, словно значение указателя p изменилось, на самом деле это не так. Напомним, что указатель является адресом ячейки памяти, где хранится значение определенного типа. После вызова функции sneaky адрес, хранящийся в переменной p, остался неизменным — изменилось только значение, хранящееся в памяти по этому адресу. Если параметр функции имеет тип класса или структуры с переменными-членами типа указателей и передается по значению, вызов функции может стать причиной таких же неожиданных изменений. Однако для типа класса эти изменения можно предотвратить, объявив специальный *конструктор копирования*, рассматриваемый в следующем разделе.

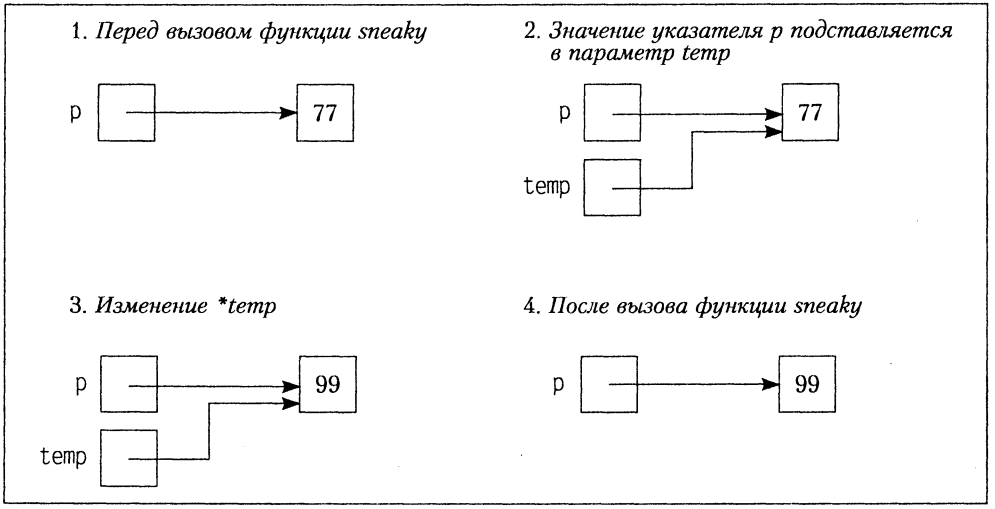


Рис. 12.3. Вызов функции sneaky

Конструктор копирования

Конструктор копирования — это конструктор, имеющий один параметр типа класса, к которому относится данный конструктор. Такой параметр должен передаваться по ссылке, и обычно ему предшествует квалификатор const, объявляющий его константным. Во всем остальном конструктор копирования определяется и применяется так же, как любой другой конструктор.

Например, программа, использующая класс StringVar, определенный в листинге 12.5, может содержать следующий код:

```
StringVar line(20), motto("Constructors can help.");
cout << "Enter a string of length 20 or less:\n";
line.input_line(cin);
StringVar temp(line);   // Инициализируется конструктором копирования.
```

Конструктор, который применяется для инициализации каждого из трех объектов типа StringVar, определяется типом аргумента, заданного в круглых скобках после имени объекта. Так, объект line инициализируется конструктором с параметром

типа int, а объект motto — конструктором с параметром типа const char a[]. Аналогично объект temp инициализируется конструктором с одним аргументом типа const StringVar&. В данном случае конструктор копирования используется так же, как любой другой конструктор.

Конструктор копирования должен быть определен таким образом, чтобы инициализируемый им объект становился полной независимой копией аргумента. Поэтому объявление

```
StringVar temp(line);
```

не просто присваивает переменной-члену temp.value такое же значение, как у переменной-члена line.value; этим бы создавались два указателя на один и тот же динамический массив. Данное объявление создает объект с указателем на другой динамический массив в памяти, содержащий точно такие же данные, как массив, на который указывает line.value. Определение конструктора копирования класса StringVar приведено в листинге 12.7. Обратите внимание: в нем создается новый динамический массив, куда копируется содержимое другого динамического массива. Таким образом, в приведенном выше объявлении объект temp инициализируется так, что его строковое значение становится равным строковому значению объекта line, но при этом он содержит отдельный динамический массив, благодаря чему любые изменения в объекте temp не отражаются на объекте line.

Мы уже говорили, что конструктор копирования может использоваться так же, как любой другой конструктор. Более того, в определенных ситуациях он вызывается автоматически. (В частности, когда компилятору C++ требуется копия объекта, он автоматически вызывает конструктор копирования.)

Конструктор копирования вызывается автоматически когда:

- объект класса объявляется и инициализируется другим объектом того же типа;
- функция возвращает значение типа класса;
- аргумент типа класса подставляется в параметр, передаваемый по значению. В данном случае конструктор копирования определяет объект, подставляемый в параметр.

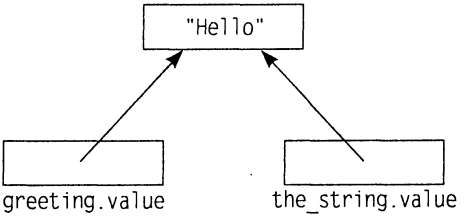
Чтобы понять, зачем нужен конструктор копирования, посмотрим, что произойдет, если не определить его для класса StringVar. Предположим, определение класса StringVar не содержит конструктор копирования и при этом в определении функции объявлен параметр, предаваемый по значению, например:

```
void show_string(StringVar the_string)
{
    cout << "The string is: "
         << the_string << endl;
}
```

Рассмотрим следующий код, содержащий вызов функции:

```
StringVar greeting("Hello");
show_string(greeting);
cout << "After call: " << greeting << endl;
```

В случае отсутствия конструктора копирования выполнение данного кода происходит следующим образом. Когда вызывается функция show_string, значение объекта greeting копируется в ее локальную переменную the_string, так что значение переменной-члена the_string.value устанавливается равным значению greeting.value. Однако эти переменные являются указателями, и поэтому во время выполнения функции show_string обе они указывают на один и тот же динамический массив, вот так:



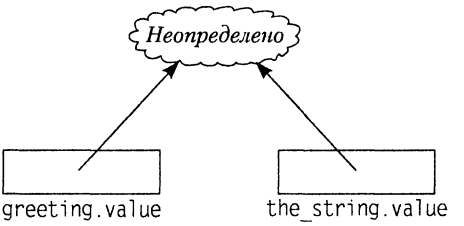
После завершения выполнения функции вызывается деструктор класса StringVar, возвращающий занимаемую объектом the_string память в динамическую область. В определении деструктора имеется следующий оператор:

```
delete [] value;
```

Поскольку деструктор вызывается для объекта the_string, этот оператор эквивалентен оператору

```
delete [] the_string.value;
```

Когда заканчивается его выполнение, «картина» становится такой:



Поскольку переменные greeting.value и the_string.value указывают на один и тот же динамический массив, удаление второй из них означает и удаление первой. В результате по достижении программой следующего оператора значение переменной greeting.value оказывается не определенным:

```
cout << "After call: " << greeting << endl;
```

В итоге действие потока cout тоже становится не определенным. Возможно, будет выведено то, что ожидалось, но рано или поздно не определенное значение greeting.value приведет к проблемам. Одна из них возникает, когда объект greeting является локальной переменной некоторой функции. В этом случае по завершении

работы данной функции для него вызывается деструктор, вызов которого эквивалентен выполнению следующего оператора:

```
delete [] greeting.value;
```

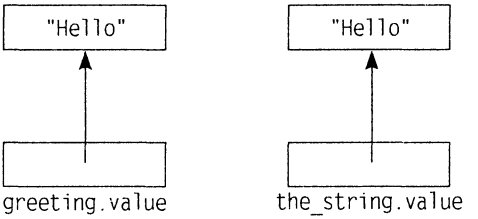
Но как мы только что видели, динамический массив, на который указывает greeting.value, один раз уже удален, и теперь система пытается удалить его во второй раз. Повторный вызов оператора delete для одного и того же динамического массива (или другой переменной, созданной с помощью оператора new) может привести к серьезной системной ошибке и сбою программы.

Все это произойдет, если у класса StringVar не окажется конструктора копирования. К счастью, мы включили в его определение такой конструктор, и при выполнении следующего вызова:

```
StringVar greeting("Hello");
show_string(greeting);
```

он будет вызываться автоматически.

Конструктор копирования определяет объект, который подставляется в параметр the_string вместо аргумента greeting, и указатели теперь ссылаются так:



Таким образом, никакие изменения в значении переменной-члена the_string.value больше не отражаются на аргументе greeting, и с деструктором тоже нет никаких проблем. Если деструктор вызывается для объекта the_string, а потом для объекта greeting, то эти два вызова удаляют разные динамические массивы.

Когда функция возвращает значение типа класса, конструктор копирования вызывается автоматически для копирования значения, заданного в операторе return. И если такой конструктор не определен, возникают такие же проблемы, как при передаче параметров по значению.

Если определение класса содержит указатели и для объектов этого класса с помощью оператора new выделяется динамическая память, то такому классу требуется конструктор копирования. В противном случае (то есть если класс не работает с динамически выделяемой памятью) этот конструктор ему не нужен.

Обратите внимание, что конструктор копирования не вызывается, когда один объект присваивается другому с помощью оператора присваивания. Но если вас не устраивают действия стандартного оператора присваивания, вы имеете возможность переопределить его так, как описано далее в разделе «Перегрузка оператора присваивания».

Конструктор копирования

Конструктор копирования — это конструктор с одним передаваемым по ссылке параметром типа класса, к которому относится этот конструктор. Обратите особое внимание, что параметр должен передаваться по ссылке; кроме того, обычно его объявляют константным, то есть ставят перед ним квалификатор const. Конструктор копирования класса вызывается автоматически, когда какая-либо функция возвращает значения типа этого класса. Кроме того, он вызывается при передаче по значению аргумента типа этого класса в функцию. Помимо этого, он может использоваться в программах так же, как другие конструкторы класса.

Каждый класс, в котором применяются указатели и оператор new, должен содержать конструктор копирования.

Большая тройка

Конструктор копирования, оператор = и деструктор называются *большой тройкой*, поскольку профессионалы утверждают, что в тех случаях, когда классу требуется один из них, следует определять все три. Если одна из этих функций-членов класса не определена, компилятор создает ее сам, но в его реализации она может вести себя не так, как требуется программе. Поэтому лучше определять эти функции самостоятельно. Сгенерированные компилятором конструктор копирования и перегруженный оператор = будут работать успешно, если все переменные-члены класса относятся к предопределенным типам, таким как int или double. Но если класс содержит переменные-члены типа классов или указателей, они могут работать не так, как следует. Для любого класса, в котором используются указатели и оператор new, следует определить собственный конструктор копирования, перегруженный оператор = и деструктор.

Упражнения для самопроверки

16. Если класс имеет имя MyClass и у него есть конструктор, каково имя этого конструктора? Каково имя деструктора, если он имеется у данного класс?

17. Предположим, что вы следующим образом изменили определение деструктора, приведенное в листинге 12.7:

```
StringVar::~StringVar()
{
    cout << endl
         << "Good-bye cruel world! The short life of\n"
         << "this dynamic array is about to end.\n";
    delete [] value;
}
```

    Как при этом изменится пример диалога из листинга 12.6?

18. Ниже приведена первая строка определения конструктора копии для класса StringVar.

```
StringVar::StringVar(const StringVar& string_object)
```

    Здесь трижды встречается идентификатор StringVar, причем все три вхождения имеют разное значение. Опишите эти значения.

19. Ответьте на следующие вопросы о деструкторах:
    а) что такое деструктор, и каким должно быть его имя;
    б) когда вызывается деструктор;
    в) что делает деструктор на самом деле;
    г) что он должен делать?

Перегрузка оператора присваивания

Предположим, что переменные string1 и string2 объявлены таким образом:

```
StringVar string1(10), string2(20);
```

Класс StringVar определен в листингах 12.5 и 12.7. Если переменной string2 присвоено значение, то следующий оператор присваивания:

```
string1 = string2;
```

допустим, но его действие может быть не таким, как хотелось бы.

Как обычно, эта предопределенная версия оператора присваивания копирует значение каждой переменной-члена объекта string2 в соответствующую переменную-член объекта string1, так что значение переменной-члена string1.max_length оказывается таким же, как значение переменной-члена string2.max_length, а значение string1.value — таким же, как значение string2.value. Однако в программном коде это может вызвать проблемы не только при работе с переменной string1, но даже и при обработке переменной string2.

Переменная-член string1.value содержит указатель, и оператор присваивания устанавливает его равным значению string2.value. В результате оба эти указателя ссылаются на одну и ту же ячейку памяти. Если изменить строковое значение в string1, оно изменится также в string2, и наоборот.

Короче говоря, предопределенный оператор присваивания делает не то, что следовало бы делать для объектов типа StringVar. Использование с классом StringVar предопределенной версии этого оператора может стать источником серьезных проблем в программе. Для того чтобы исправить ситуацию, программист должен самостоятельно перегрузить оператор присваивания, чтобы он выполнял с объектами класса те действия, которые соответствуют их структуре и назначению.

Оператор присваивания нельзя перегружать тем же способом, что другие операторы, такие как << и +. Его перегруженная версия должна быть не дружественной функцией класса, а его членом. Чтобы добавить в класс StringVar перегруженную версию оператора присваивания, нужно изменить определение этого класса следующим образом:

```
class StringVar
{
public:
    void operator =(const StringVar& right_side);
    // Перегружает оператор присваивания = для копирования
    // строки из одного объекта в другой.
    ... Остальная часть определения класса такая же,
        как в листинге 12.5. ...
```

После этого (и после добавления его определения) оператор присваивания используется совершенно так же, как обычно. В качестве примера рассмотрим следующую строку кода:

```
string1 = string2;
```

Здесь string1 является вызывающим объектом функции-члена, реализующей оператор присваивания, а string2 — ее аргументом.

Определение нового оператора присваивания таково:

```
// Данное определение допустимо, но далее
// приводится его улучшенная версия:
void StringVar::operator =(const StringVar& right_side)
{
    int new_length = strlen(right_side.value);
    if ((new_length) > max_length)
        new_length = max_length;
    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
        value[new_length] = '\0';
}
```

Обратите внимание на проверку длины строки в объекте с помощью оператора if. Если она слишком велика для объекта, заданного слева от оператора присваивания (то есть для вызывающего объекта), то в него копируется такое количество символов, которое он может вместить. Но предположим, мы не хотим, чтобы в результате копирования терялась часть символов. При копировании всей строки нужно создать для принимающего объекта новый динамический массив большего размера. Можно попробовать переопределить оператор присваивания следующим образом:

```
// В этой версии имеется ошибка.
void StringVar::operator =(const StringVar& right_side)
{
    delete [] value;
    int new_length = strlen(right_side.value);
    max_length = new_length;

    value = new char[max_length + 1];

    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
        value[new_length] = '\0';
}
```

Если с двух сторон от оператора присваивания задать один и тот же объект:

```
my_string = my_string;
```

при его выполнении первым делом будет выполнен оператор

```
delete [] value;
```

Однако вызывающим объектом является my_string, так что данный оператор эквивалентен следующему:

```
delete [] my_string.value;
```

Строковое значение объекта my_string удаляется, и указатель my_string.value оказывается не определенным. Таким образом, оператор присваивания просто разрушил объект my_string, что, конечно, не способствует правильному продолжению работы программы.

Для того чтобы исправить эту ошибку, нужно прежде всего проверить, достаточно ли места в динамическом массиве-члене объекта, заданного справа от оператора присваивания, и удалять массив только в том случае, если места недостаточно. Эту проверку выполняет наша окончательная версия перегруженного оператора присваивания:

```
// Это окончательная версия.
void StringVar::operator =(const StringVar& right_side)
{
    int new_length = strlen(right_side.value);
    if (new_length > max_length)
    {
        delete [] value;
        max_length = new_length;
        value = new char[max_length + 1];
    }
    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}
```

Для многих классов самое очевидное определение оператора присваивания неправильно работает в том случае, если с двух сторон от него задан один и тот же объект. Поэтому всегда проверяйте этот случай и учитывайте его при написании определения перегруженного оператора присваивания.

Упражнение для самопроверки

20. Объясните подробно, почему перегруженный оператор присваивания не нужен в тех случаях, когда класс содержит только переменные встроенных типов данных. Объясните то же самое для конструктора копирования и для деструктора.

Резюме

- Указатель — это адрес переменной в памяти, его использование является способом неявного именования переменной путем указания имени ее адреса в памяти компьютера.
- Динамическая переменная — это переменная, создаваемая и удаляемая во время выполнения программы.
- Память для динамических переменных выделается в особой области компьютерной памяти, называемой динамической. Когда программа завершает работу с динамической переменной, занятую этой переменной память необходимо вернуть в динамическую область для повторного использования; эта операция выполняется с помощью оператора delete.

- Динамический массив — это массив, размер которого определяется во время выполнения программы. Динамический массив реализуется с помощью динамической переменной типа массива.
- Деструктор представляет собой особый вид функции-члена класса. Он вызывается автоматически, когда объект класса выходит из области видимости программы. Его основная задача — вернуть в динамическую область память, занимаемую динамическими данными объекта.
- Конструктор копирования является конструктором с единственным аргументом — объектом класса, к которому относится данный конструктор. Если конструктор копирования определен, он автоматически вызывается, когда какая-либо функция возвращает значение типа класса или принимает по значению аргумент типа класса. Конструктор копирования должен быть у каждого класса, использующего указатели и оператор new.
- Оператор присваивания = можно перегрузить для класса, чтобы он выполнял свою задачу так, как требуется этому классу. Однако данный оператор необходимо перегружать как член класса, а не как его дружественную функцию. Каждый класс, использующий указатели и оператор new, должен содержать перегруженный оператор присваивания.

Ответы к упражнениям для самопроверки

1. Указатель — это адрес переменной в памяти.
2. Для новичка оно выглядит так, словно объявляются два указателя на тип данных int. Однако символ звездочки относится к идентификатору переменной, а не к имени типа (то есть к int_ptr1, а не к int). Поэтому данный оператор объявляет переменную int_ptr1 как указатель на int и переменную int_ptr2 как обычную переменную типа int.
3.
```
int *p;  // Объявляется переменная для хранения
         // указателя на переменную типа int.
*p = 17; // Здесь символ звездочки является оператором разыменования.
         // Данный оператор помещает по адресу
         // переменной p значение 17.
```
4.
```
10 20
20 20
30 30
```
   Если заменить оператор *p1 = 30; оператором *p2 = 30;, программа выведет то же самое.
5.
```
10 20
20 20
30 20
```
6.
```
delete p;
```

7.
```
typedef int* NumberPtr;
NumberPtr my_point;
```

8. Оператор new принимает в качестве аргумента имя типа. Он выделяет в динамической области память для переменной данного типа и возвращает указатель на эту память (то есть указатель на новую динамическую переменную), конечно, если в динамической области хватает места. Если же места в динамической области недостаточно, работа программы завершается.

9.
```
typedef char* CharArray;
```

10.
```
cout << "Enter 10 integers:\n";
for (int i = 0; i < 10; i++)
    cin >> entry[i];
```

11.
```
delete [] entry;
```

12. 0 1 2 3 4 5 6 7 8 9

13. 10 1 2 3 4 5 6 7 8 9

14. 0 1 2 3 4 5 6 7 8 9

15. 1 2 3 4 5 6 7 8 9

16. Конструктор называется MyClass — так же, как класс, к которому он принадлежит. Деструктор именуется ~MyClass.

17. Диалог изменится следующим образом:

```
What is your name?
Kathryn Janeway
We are Borg
We will meet again Kathryn Janeway

Good-bye cruel world! The short life of
this dynamic array is about to end.

Good-bye cruel world! The short life of
this dynamic array is about to end.
End of demonstration
```

18. StringVar перед символом :: — это имя класса, а сразу после символа :: — имя функции-члена (напомним, что конструктор является функцией-членом класса, имя которой совпадает с именем класса), StringVar в скобках — это тип параметра string_object.

19. а) деструктор является функцией-членом класса. Его имя всегда начинается с символа тильды (~), за которым следует имя класса;

    б) деструктор вызывается, когда объект класса выходит из области видимости программы;

    в) деструктор делает то, что захочет автор класса;

    г) деструктор предназначен для удаления созданных объектом класса динамических переменных. Кроме того, он может выполнять и другие задачи, связанные с удалением объекта.

20. Оператор присваивания и конструктор копирования не нужны, потому что используемый по умолчанию механизм копирования для стандартных типов работает прекрасно. Деструктор же не нужен, потому что такой класс не выполняет динамического выделения памяти и ее не нужно возвращать системе.

Практические задания

1. Дополните определение класса StringVar, приведенное в листингах 12.5 и 12.7, следующими элементами:
   - функцией-членом copy_piece, возвращающей заданную подстроку; функцией-членом one_char, возвращающей заданный символ, и функцией-членом set_char, изменяющей заданный символ;
   - перегруженной версией оператора == (она должна сравнивать только строковые значения, а значения переменной-члена max_length не должны совпадать);
   - перегруженной версией оператора +, выполняющей конкатенацию строк типа StringVar;
   - перегруженной версией оператора ввода, считывающей одно слово (в отличие от функции input_line, считывающей целую строку).

   Если вы внимательно прочли раздел о перегрузке оператора присваивания, добавьте в класс этот оператор. Напишите подходящую отладочную программу и тщательно протестируйте определение класса.
2. Выполните практическое задание 10 главы 10 с использованием динамического массива. В этой версии класса один из конструкторов должен иметь единственный аргумент типа int, определяющий максимальное количество элементов списка.
3. Выполните практическое задание 9 из главы 10 с использованием динамического массива. Программа должна запрашивать у пользователя количество чеков каждой категории и применять эту информацию для определения размера динамических массивов.
4. В главе 11 рассказывалось о векторах — структурах данных, подобных массивам, размер которых может увеличиваться во время выполнения программы. Предположим, что векторы не определены в C++. Определите класс VectorDouble, похожий на класс векторов базового типа double. У этого класса должна быть переменная-член для динамического массива значений типа double. Кроме того, он должен содержать две переменные типа int: max_count для хранения размера динамического массива и count для количества заполненных в данный момент элементов. (Первая переменная-член является аналогом емкости вектора, а вторая — аналогом его размера).

   Если при попытке добавления элемента (значения типа double) в объект-вектор класса VectorDouble для него окажется не достаточно места, класс должен создать новый динамический массив, с удвоенной по сравнению с исходной емкостью, и скопировать в него значения из исходного массива.

Класс должен содержать следующие элементы:

- три конструктора: используемый по умолчанию конструктор, создающий динамический массив из 50 элементов, конструктор с одним аргументом типа int для первоначального количества элементов в динамическом массиве и конструктор копирования;
- деструктор;
- подходящий перегруженный оператор присваивания (=);
- подходящий перегруженный оператор равенства (==). Для того чтобы объекты класса VectorDouble считались равными, должны быть равны значения их переменных-членов count и значения count элементов массивов; равенство значений переменных-членов max_count не требуется;
- функции-члены push_back, capacity, size, reserve и resize, которые должны вести себя так же, как одноименные функции-члены вектора;
- две функции-члена, выполняющие ту же задачу, что и квадратные скобки для вектора: value_at(i), возвращающая значение i-го элемента динамического массива, и change_value_at(d, i), присваивающая i-му элементу динамического массива значение d типа double. Наложите на значения аргументов этих функций подходящие ограничения. (Класс не должен работать с квадратными скобками; это можно сделать, но в нашей книге данная тема не рассматривается.)

5. Определите класс Text, в объектах которого хранятся списки слов. Этот класс подобен классу StringVar, но базовым типом данных его динамического массива является сам класс StringVar, а не тип char, и конец списка значений типа StringVar отмечается объектом StringVar, содержащим один пробел, а не символом '\0'. Объект класса Text представляет некоторый текст, состоящий из разделенных пробелами слов. Элементы массива типа StringVar не должны содержать пробелов (за исключением последнего элемента, выполняющего роль маркера конца списка).

   У класса Text должны быть функции-члены, соответствующие всем функциям-членам класса StringVar. Конструктор с аргументом типа const char a[] инициализирует объект Text так же, как описанная ниже функция input_line. Если аргумент типа строки С содержит символ перевода строки ('\n'), это считается ошибкой и работа программы завершается выводом соответствующего сообщения.

   Функция-член input_line считывает строки, разделенные пробелами, и записывает каждую из них в один элемент динамического массива базового типа StringVar. Последовательности из нескольких пробелов интерпретируются как один пробел. При выводе объекта класса Text вставляйте между каждой парой значений типа StringVar пробел. Можете считать, что символы табуляции не используются, или же интерпретировать их как пробелы.

   Дополните класс Text так, как указано в описании практического задания 1. Перегруженная версия оператора ввода >> должна заполнять только один элемент динамического массива.

6. Используя динамические массивы, реализуйте класс для представления многочленов с операциями сложения, вычитания и умножения.

   Примечание. Переменная многочлена не имеет самостоятельного значения. Поэтому его можно представить в виде массива коэффициентов и соответствующих значений степени. Рассмотрим многочлен:

   ```
   x*x*x + x + 1
   ```

   При реализации класса многочленов проще всего определить массив значений типа `double` для коэффициентов, а индексы этого массива использовать в качестве значений степеней соответствующих членов многочлена. Где же в этом массиве будет находиться член `x*x` для приведенного выше примера? Если данный член отсутствует в многочлене, это просто означает, что его коэффициент равен нулю.

   Существуют специальные технологии представления многочленов с высокими степенями и большим количеством пропущенных членов (такие многочлены называют разреженными). Однако вам не обязательно пользоваться этими технологиями.

   Определите для создаваемого класса конструктор, применяемый по умолчанию, конструктор копирования и конструктор с параметрами, позволяющий составлять произвольные многочлены. Кроме того, задайте перегруженный оператор = и деструктор.

   Определите следующие операции:

   многочлен + многочлен;

   константа + многочлен;

   многочлен + константа;

   многочлен – многочлен;

   константа – многочлен;

   многочлен – константа;

   многочлен * многочлен;

   константа * многочлен;

   многочлен * константа.

   Определите функции для присваивания и извлечения коэффициентов, индексируемых степенями, а также функцию для вычисления значения многочлена как значения типа `double`.

   Решите самостоятельно, как объявлять эти функции: как члены класса, дружественные функции класса или независимые функции.

Глава 13 Рекурсия

После лекции о космологии и строении солнечной системы
к Уильяму Джеймсу подошла пожилая леди.
— Ваша теория о том, что солнце находится в центре солнечной системы,
а земля является вращающимся вокруг нее шаром, очень убедительна,
мистер Джеймс, но она не верна.
У меня есть лучшая теория, — сказала она.
— Что же это за теория? — вежливо поинтересовался Джеймс.
— Мы живем на большой земляной корке на спине гигантской черепахи.
Не желая опровергать эту абсурдную теорию
имеющимися у него научными доказательствами,
Джеймс решил тактично разубедить своего оппонента
и помочь ей самой увидеть противоречивость ее идеи.
— Если ваша теория верна, мадам, — спросил он, — на чем же стоит эта черепаха?
— О, вы очень умны, мистер Джеймс, и это прекрасный вопрос, — ответила
пожилая леди, — но у меня есть на него ответ.
Вот он: черепаха стоит на спине другой, еще большей черепахи.
— Но на чем же стоит эта вторая черепаха? — терпеливо продолжал Джеймс.
На это пожилая леди торжествующе воскликнула:
— Не выйдет, мистер Джеймс — там, внизу, только черепахи — одна
под другой все ниже и ниже!

Дж. Росс

Вам уже встречалось несколько примеров циклических определений. Наиболее наглядными из них являются определения некоторых операторов C++. Например, в описании оператора `while` сказано, что он может содержать другие операторы. Поскольку в частном случае это может быть и оператор `while`, определение получается циклическим. В математике такие циклические определения называются *рекурсивными*. В C++ подобным образом можно определять функции. Точнее говоря, определение функции может содержать вызов этой же функции. Такая функция именуется *рекурсивной*. Данная глава посвящена рекурсии в языке C++, а также более общему понятию рекурсии как технологии программирования и решения задач.

13.1. Рекурсивные функции, не возвращающие значений

Вспомнил я и ту ночь посередине «Тысячи и одной ночи», когда Шехерезада по чудесной оплошности переписчика принимается дословно пересказывать историю «Тысячи и одной ночи», рискуя вновь добраться до той ночи, когда она ее пересказывает, и так до бесконечности.

Хорхе Луис Борхес

Создавая функцию для решения какой-либо задачи, обычно стараются разбить такую задачу на меньшие подзадачи. При этом может оказаться, что одна из подзадач представляет собой сокращенную версию исходной задачи. Если задача состоит в том, чтобы найти в массиве конкретное значение, можно разделить ее на две подзадачи: поиск этого значения отдельно в первой и во второй частях массива. Во всех подобных случаях можно решить исходную задачу с использованием рекурсивной функции. Перед ее применением придется немного потренироваться, но в результате у вас в руках окажется один из эффективнейших способов разработки алгоритмов, а следовательно, и функций C++. Начнем с простого примера.

Пример: вертикальная запись чисел

В этом примере мы разработаем рекурсивную функцию типа `void`, записывающую числа на экране вертикально, а не горизонтально, так что выведенное ею число 1984 будет выглядеть следующим образом:

```
1
9
8
4
```

Постановка задачи

Объявление и поясняющий комментарий функции таковы:

```
void write_vertical(int n);
// Предусловие: n >= 0.
// Постусловие: число n выводится на экран вертикально,
// каждая цифра – с новой строки.
```

Разработка алгоритма

Первый вариант входных данных очень прост. Если подлежащее выводу на экран число `n` состоит из единственной цифры, нужно просто вывести его как есть. Хотя это и простейший случай, он важен для нашего примера, поэтому опишем его:

```
Простейший случай: если n < 10, тогда вывести число n на экран.
```

Теперь рассмотрим более типичный случай, когда число содержит больше, чем одну цифру. Так, если нужно вывести число 1234, результат будет следующим:

```
1
2
3
4
```

Разделить эту задачу на подзадачи можно описанным ниже образом.

1. Вывести цифры за исключением последней:

   1
   2
   3

2. Вывести последнюю цифру, в этом примере 4.

Подзадача 1 является сокращенной версией исходной задачи, так что ее можно реализовать в виде рекурсивного вызова. А подзадача 2 — это описанный выше простейший случай. Поэтому определение нашего алгоритма для функции write_vertical с параметром n можно записать в виде следующего псевдокода:

```
if (n < 10)
{
    cout << n << endl;
}
else   // n имеет длину 2 или более цифры.
{ // Рекурсивная подзадача.
    write_vertical(число n без последней цифры);
    cout << последняя цифра n << endl;
}
```

Чтобы превратить данный псевдокод в функцию, нужно преобразовать в выражения C++ следующие два фрагмента:

```
число n без последней цифры
```

и

```
последняя цифра n
```

Это легко сделать с помощью операторов целочисленного деления и вычисления остатка:

```
n/10   // Число n без последней цифры.
n%10   // Последняя цифра n.
```

Например, оператор 1234/10 возвращает 123, а 1234%10 — 4.

Выбор именно таких подзадач при анализе нашего алгоритма произведен не случайно, он определяется двумя факторами.

Первый из них состоит в том, что аргумент рекурсивного вызова функции write_vertical, выделенного в приведенном выше псевдокоде, вычислить несложно. Для получения числа n без последней цифры достаточно выполнить оператор n/10. В качестве альтернативы можно было бы попробовать определить подзадачи иным образом.

1 Вывод первой цифры числа n.

2. Вывод числа n без первой цифры.

Это вполне допустимая декомпозиция задачи, и ее можно реализовать рекурсивно. Однако результат удаления первой цифры числа вычислить труднее, чем результат удаления последней цифры.

Вторым фактором является то, что одна из подзадач не требует рекурсивного вызова. Последовательное определение рекурсивной функции всегда включает хотя

бы одно условие, не требующее рекурсивного вызова (и одно или более условий, требующих хотя бы одного рекурсивного вызова), но об этом мы поговорим в разделе, следующем за данным примером.

Написание программного кода

Теперь можно собрать все разработанные выше элементы алгоритма в единую функцию (листинг 13.1). В следующем разделе принцип действия рекурсии в нашем примере рассматривается подробнее.

Процесс выполнения рекурсивных вызовов

Посмотрим, что же происходит при выполнении такого вызова:

```
write_vertical(123);
```

Листинг 13.1. Рекурсивная функция для вывода значения

```
// Программа, демонстрирующая рекурсивную функцию write_vertical.
#include <iostream>
using namespace std;

void write_vertical(int n);
// Предусловие: n >= 0.
// Постусловие: число n выводится на экран вертикально,
// каждая цифра – с новой строки.


int main()
{
    cout << "write_vertical(3):" << endl;
    write_vertical(3);

    cout << "write_vertical(12):" << endl;
    write_vertical(12);

    cout << "write_vertical(123):" << endl;
    write_vertical(123);

    return 0;
}

// Используем библиотеку классов iostream.
void write_vertical(int n)
{
    if (n < 10)
    {
        cout << n << endl;
    }
    else   // n имеет длину 2 или более цифры:
    {
        write_vertical(n/10);
        cout << (n%10) << endl;
    }
}
```

Пример диалога

```
write_vertical(3):
3
write_vertical(12):
1
2
write_vertical(123):
1
2
3
```

Выполняя очередной вызов функции write_vertical, компьютер действует так же, как при вызове любой другой функции. Аргумент 123 подставляется в параметр n в определении функции, а затем выполняется ее тело. После подстановки этого значения выполнение кода функции происходит так:

```
if (123 < 10)
{
    cout << 123 << endl;
}
else  // n имеет длину 2 или более цифры:
{
    write_vertical(123/10);
    cout << (123%10) << endl;
}
```

Здесь вычисления приостановятся, и будет выполнен рекурсивный вызов

Поскольку 123 не меньше 10, логическое выражение в операторе if...else возвращает false и выполняется блок else. Однако этот блок начинается со следующего вызова:

```
write_vertical(n/10);
```

равнозначного (поскольку n равен 123) вызову:

```
write_vertical(123/10);
```

который, в свою очередь, эквивалентен такому:

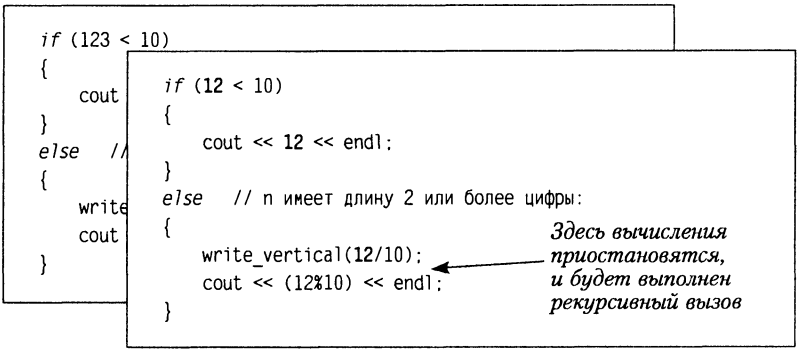
```
write_vertical(12);
```

Когда выполнение функции достигает рекурсивного вызова, текущие вычисления приостанавливаются и выполняется этот вызов. После его завершения вычисления возобновляются.

Рекурсивный вызов

```
write_vertical(12);
```

обрабатывается так же, как вызов любой другой функции. Аргумент 12 подставляется в параметр n, после чего выполняется тело функции. После указанной подстановки в программе будут содержаться уже два процесса вычислений — один приостановленный, а второй активный.



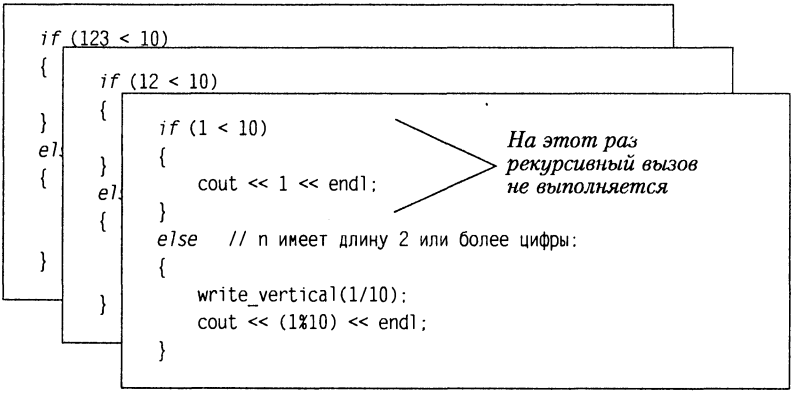
Поскольку 12 не меньше 10, логическое выражение в операторе `if...else` возвращает `false` и выполняется блок `else`. Однако, как вы уже видели, этот блок начинается с рекурсивного вызова. Аргумент рекурсивного вызова, `n/10`, на этот раз равен 12/10. Таким образом, второе выполнение функции `write_vertical` тоже приостанавливается и выполняется следующий вызов:

```
write_vertical(12/10);
```

эквивалентный такому:

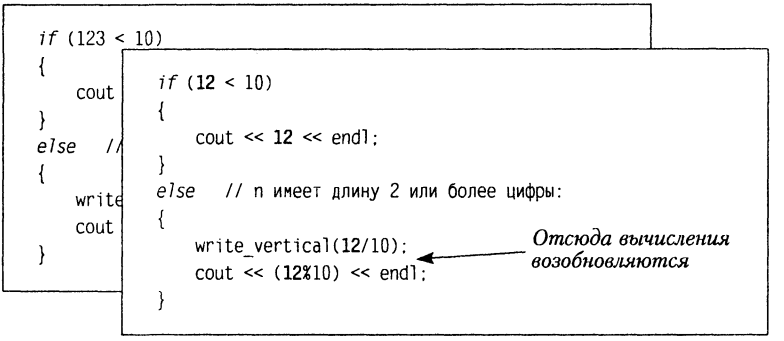
```
write_vertical(1);
```

В этот момент в программе уже два процесса вычислений ожидают возобновления, и компьютер начинает выполнение нового рекурсивного вызова, который обрабатывается так же, как предыдущие. Аргумент 1 подставляется в параметр `n`, после чего выполняется тело функции. Теперь выполнение программы выглядит следующим образом.

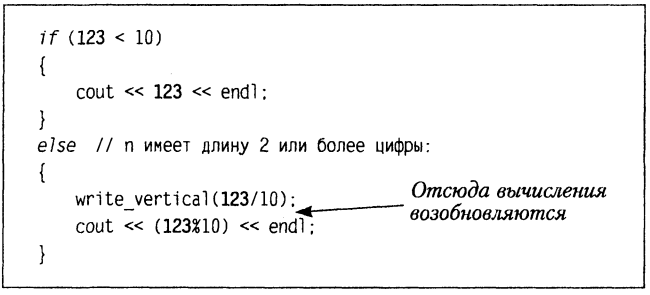


На этот раз выполнение тела функции происходит иначе. Поскольку 1 меньше 10, логическое выражение в операторе `if...else` возвращает значение `true` и выполняется блок `if`. Он состоит из единственного объекта `cout`, выводящего на экран 1, так что вызов `write_vertical(1)` выводит на экран единицу и завершается без дальнейших рекурсивных вызовов.

После этого приостановленные вычисления второго вызова функции write_vertical возобновляются с той точки, на которой они были остановлены, как показано на следующем рисунке.



После возобновления приостановленных вычислений обрабатывается объект cout, который выводит значение выражения 12%10, то есть 2. На этом работа данного вызова функции завершается, и возобновляются последние приостановленные вычисления.



Когда и эти последние (но начавшиеся первыми) вычисления возобновляются, программа выводит значение выражения 123%10, равное 3, и выполнение исходного вызова функции write_vertical заканчивается. Теперь на экран последовательно выведены цифры 1, 2 и 3, причем каждая с новой строки.

Подробно о рекурсии

В определении функции write_vertical используется рекурсия. Однако, разбирая вызов этой функции, write_vertical(123), мы не увидели в нем ничего нового. Он выполнялся точно так же, как вызовы других функций, рассмотренных в предыдущих главах. Аргумент 123 просто подставлялся в параметр n, а затем выполнялся код тела определения функции. По достижении рекурсивного вызова

```
write_vertical(123/10);
```

этот процесс повторялся еще один раз.

Компилятор отслеживает рекурсивные вызовы следующим образом. После вызова функции он подставляет аргументы в ее параметры и начинает выполнять код. Встретив рекурсивный вызов, компилятор временно останавливает выполнение кода, поскольку для его продолжения нужно получить результат рекурсивного вызова. Компьютер сохраняет информацию, необходимую для последующего продолжения обработки функции, и переходит к выполнению рекурсивного вызова. Когда этот вызов завершается, прерванные вычисления внешнего вызова возобновляются.

Язык C++ не ограничивает использование рекурсивных вызовов в определениях функций. Но для создания эффективного рекурсивного определения его следует разрабатывать так, чтобы рано или поздно такие вызовы завершались выполнением кода без рекурсии. Общая схема успешного определения рекурсивной функции такова.

- При наличии одного или нескольких условий функция выполняет свою задачу с использованием рекурсивных вызовов, соответствующих одной или нескольким сокращенным версиям задачи.
- При наличии одного или нескольких условий функция выполняет свою задачу без использования рекурсивных вызовов. Эти условия называются *базовыми* или *условиями останова*.

Очень часто условие, от которого зависит выполнение следующего рекурсивного вызова, задается в операторе `if...else`. Согласно типичной схеме в обычном вызове функции возможны условия, при которых требуется рекурсивный вызов. В этом вызове, в свою очередь, возможны условия, при которых потребуется еще один рекурсивный вызов. Некоторое количество раз каждый рекурсивный вызов будет выполнять следующий рекурсивный вызов, но в какой-то момент наступит условие останова. В каждом вызове функции со временем должно сформироваться условие останова — в противном случае вызов никогда не завершится из-за бесконечной цепочки рекурсивных вызовов. (На практике бесконечная цепочка рекурсивных вызовов рано или поздно приведет к аварийному завершению работы программы.)

Самый распространенный способ обеспечить наступление условий останова — написать функцию таким образом, чтобы в каждом ее вызове уменьшалось некоторое положительное числовое значение и условие останова выполнялось при некоторой достаточно малой величине этого значения. Именно так спроектирована функция `write_vertical`, приведенная в листинге 13.1. Ее рекурсивные вызовы каждый раз выполняются с меньшим значением аргумента, и когда это значение оказывается меньше, чем `10`, рекурсивные вызовы прекращаются, а каждый из вложенных вызовов выполняется до конца, пока не завершится исходный вызов функции.

Ловушка: бесконечная рекурсия

В приведенном ранее примере функции `write_vertical` рекурсивные вызовы выполняются до тех пор, пока в одном из вызовов не будут содержаться такие данные, что рекурсия больше не требуется (то есть пока не будет выполнено условие

останова). Но если каждый рекурсивный вызов будет выполнять еще один рекурсивный вызов, то выполнение функции, по крайней мере теоретически, никогда не завершится. Это называется *бесконечной рекурсией*. На практике такая функция будет выполняться, пока не исчерпает ресурсы компьютера, после чего произойдет аварийное завершение работы программы. Иными словами, рекурсивные вызовы не должны выполняться «один за другим, все дальше и дальше» по определению пожилой леди, описывавшей строение вселенной (в эпиграфе к настоящей главе).

Пример бесконечной рекурсии придумать нетрудно. Далее приведено синтаксически правильное определение функции на языке C++, которое могло бы стать результатом попытки написать альтернативную версию функции `write_vertical`.

```
void new_write_vertical(int n)
{
    new_write_vertical(n/10);
    cout << (n%10) << endl;
}
```

Если включить такое определение в программу, вызывающую функцию `write_vertical`, компилятор транслирует его в машинный код, который можно будет выполнить. Более того, это определение даже имеет некоторый смысл. В нем сказано, что перед выводом аргумента функция `new_write_vertical` должна сначала вывести все цифры, кроме последней, а затем вывести последнюю цифру. Но на самом деле вызов данной функции приведет к бесконечной последовательности рекурсивных вызовов. Например, если вызвать ее с аргументом 12, работа функции приостановится для выполнения рекурсивного вызова `new_write_vertical(12/10)`, то есть `new_write_vertical(1)`. Выполнение этого вызова в свою очередь приостановится для выполнения рекурсивного вызова

```
new_write_vertical(1/10);
```

то есть

```
new_write_vertical(0);
```

Ну а дальше? Дальше будет вызов `new_write_vertical(0/10)`, а именно:

```
new_write_vertical(0);
```

и снова вызов `new_write_vertical(0/10)`, и т. д. до бесконечности. Поскольку в определении функции `new_write_vertical` не предусмотрено условие останова, процесс будет продолжаться вечно (или до тех пор, пока не исчерпаются ресурсы компьютера).

Упражнения для самопроверки

1. Что выведет следующая программа:

```
#include <iostream>
using namespace std;
void cheers(int n);
int main()
{
    cheers(3);
```

```
    return 0;
}

void cheers(int n)
{
    if (n == 1)
    {
        cout << "Hurray\n";
    }
    else
    {
        cout << "Hip ";
        cheers(n-1);
    }
}
```

2. Напишите рекурсивную функцию типа `void` с одним параметром, в котором задается положительное целое число, выводящую на экран соответствующее количество символов звездочки (`'*'`) в одной строке.
3. Напишите рекурсивную функцию типа `void` с одним параметром, в котором задается положительное целое число, выводящую передаваемый ей аргумент на экран в обратном порядке. Например, задав аргумент 1234, вы увидите на экране

   4321
4. Напишите рекурсивную функцию типа `void`, принимающую единственный аргумент `n` типа `int` и последовательно выводящую целые числа 1, 2, 3, ... `n`.
5. Напишите рекурсивную функцию типа `void`, принимающую единственный аргумент `n` типа `int` и последовательно выводящую целые числа `n`, `n-1`, ... 3, 2, 1. *Подсказка.* Обратите внимание, что для превращения кода упражнения 4 в код упражнения 5 (или наоборот) достаточно поменять местами пару строк.

Стеки и рекурсия

Для отслеживания рекурсии, а также для выполнения ряда других задач в большинстве систем используется способ организации и доступа к данным, называемый *стеком*. Стек можно сравнить со стопкой бумаги. Предполагается, что помимо этой стопки существует неисчерпаемый источник чистых листов. Чтобы поместить в стек новую информацию, нужно написать ее на чистом листе и положить его сверху на стопку. Таким образом стек последовательно заполняется все новой и новой информацией.

Получение информации из стека выполняется путем очень простой процедуры. Читается запись на верхнем листе стопки, и если этот лист больше не нужен, он выбрасывается. Здесь имеется одна сложность: в каждый момент времени доступен только верхний лист стопки. Чтобы прочитать, например, надпись на третьем листе, нужно выбросить верхние два. Поскольку первым из стопки всегда берется последний положенный туда лист, стек часто называют структурой памяти типа *последним вошел — первым вышел* (Last In First Out — LIFO).

Стек очень удобен для отслеживания рекурсии. При вызове функции как бы берется чистый лист бумаги, и на нем записывается определение функции с подставленными в него значениями аргументов. Затем компьютер начинает выполнение тела функции. Когда встречается рекурсивный вызов, компьютер останавливает производимые на этом листе вычисления, для того чтобы выполнить новый вызов, и отмечает место, где он остановился. На этот же лист он записывает всю информацию, необходимую для возобновления вычислений, а затем кладет заполненный лист на вершину стопки. Для рекурсивного вызова берется новый лист, и процесс повторяется. Компьютер записывает на новом листе определение функции с подставленными значениями аргументов и начинает выполнение рекурсивного вызова. Если в нем снова встречается рекурсивный вызов, на лист записывается информация, необходимая для возобновления вычислений. Он, в свою очередь, тоже кладется на вершину стопки, после чего берется новый лист. Этот процесс проиллюстрирован выше в разделе «Процесс выполнения рекурсивных вызовов». И хотя мы пока все время говорили о стопке бумаги, а не о стеке, все сказанное очень точно отражает операции компьютера со стеком.

Описанный процесс длится до тех пор, пока один из рекурсивных вызовов функции не окончится без рекурсии. Тогда компьютер снимает со стопки верхний лист и, используя записанные на нем данные, продолжает выполнение предыдущего вызова функции. Когда его выполнение завершится, снимается следующий лист и продолжается выполнение зафиксированного на нем вызова, и т. д. до исходного вызова функции. Приступая к завершению исходного вызова, компьютер берет из стопки последний лист, относящийся к выполнению данной функции, а в стопке остаются только те листы, которые были там до вызова функции. В зависимости от количества рекурсивных вызовов и от того, как написано определение функции, стопка может то расти, то уменьшаться. Обратите внимание, что листы в стопке доступны только по принципу «последним вошел — первым вышел», но для рекурсивных вызовов иное и не требуется. Каждая приостановленная копия функции все равно ждет завершения выполнения функции, лист которой расположен в стопке непосредственно над ней.

Понятно, что в компьютере используются не стопки бумаги — это лишь аналогия, но исключительно точная и понятная. На самом деле стек представляет собой последовательность фрагментов памяти. Содержимое такого фрагмента (который мы представляли как лист) называется *записью активации*, и управляется по описанной выше схеме «последним вошел — первым вышел». Записи активации содержат не полные копии определения функции, а ссылки на ее единственную копию в известном месте памяти. Однако в них имеется достаточно информации для того, чтобы компьютер мог возобновить выполнение функции.

Стек

Стеком называется структура данных в памяти, работающая по принципу «последним вошел — первым вышел». Первым доступным или удаляемым из стека элементом всегда является элемент, помещенный в него последним. Стеки удобно использовать для отслеживания рекурсивных вызовов функций.

Ловушка: переполнение стека

Размер стека всегда имеет некоторый предел. Когда функция выполняет слишком длинную цепочку рекурсивных вызовов, она в конце концов может полностью заполнить стек записями активации, после чего следующая попытка рекурсивного вызова приведет к ошибке, называемой *переполнением стека*. Если вы получите сообщение о переполнении стека, это, скорее всего, означает, что функция выполнила слишком много рекурсивных вызовов. Одной из типичных причин переполнения стека является бесконечная рекурсия.

Рекурсия и итеративное выполнение

Рекурсия не является абсолютно необходимым способом решения определенных задач. Более того, некоторые языки программирования вообще ее не допускают, поскольку любую задачу можно выполнить и без рекурсии. Например, программа в листинге 13.2 содержит нерекурсивную версию функции, приведенной в листинге 13.1. В нерекурсивных версиях, которые также можно реализовать как рекурсивные, обычно используются циклы, поэтому их часто называют *итеративными*. Если определение функции `write_vertical`, приведенное в листинге 13.1, заменить версией из листинга 13.2, вывод программы будет тем же самым. Однако рекурсивная версия функции иногда может быть проще итеративной – как в следующем примере.

Рекурсивная функция обычно работает медленнее и ей требуется больше памяти, чем эквивалентной итеративной функции. Хотя итеративная версия функции `write_vertical` в листинге 13.2 выглядит сложнее и кажется, что она требует больше памяти и выполняет больше вычислений, чем рекурсивная версия в листинге 13.1, на самом деле с точки зрения этих двух критериев обе версии очень близки. Рекурсивная версия требует даже больше памяти и менее производительна, чем итеративная, поскольку компьютеру без конца приходится работать со стеком, отслеживая рекурсивные вызовы. Однако, поскольку система делает все это автоматически, использование рекурсии иногда упрощает задачу программиста и помогает написать более простой и понятный код. Как видно из приведенных в этой главе примеров, а также упражнений и практических заданий, в одних случаях определение рекурсии является простым и понятным, а в других удобнее применять итеративное определение функции.

Листинг 13.2. Итеративная версия функции, приведенной в листинге 13.1

```
// Используем библиотеку классов iostream.
void write_vertical(int n)
{
    int tens_in_n = 1;
    int left_end_piece = n;
    while (left_end_piece > 9)
    {
        left_end_piece = left_end_piece/10;
        tens_in_n = tens_in_n*10;
    }
```

продолжение ⇩

Листинг 13.2 (*продолжение*)

```
    // tens_in_n – это степень 10 с тем же количеством
    // цифр, что и в n. Например, если значение n равно 2345,
    // значение переменной tens_in_n равно 10 000.

    for (int power_of_10 = tens_in_n;
          power_of_10 > 0; power_of_10 = power_of_10/10)
    {
        cout << (n/power_of_10) << endl;
        n = n%power_of_10;
    }
}
```

Упражнения для самопроверки

6. Какова наиболее вероятная причина ошибки, если программа выводит сообщение о переполнении стека?
7. Напишите итеративную версию функции cheers, описанной в упражнении 1.
8. Напишите итеративную версию функции, описанной в упражнении 2.
9. Напишите итеративную версию функции, описанной в упражнении 3.
10. Опишите работу рекурсивного решения упражнения 4.
11. Опишите работу рекурсивного решения упражнения 5.

13.2. Рекурсивные функции, возвращающие значения

Схема рекурсивных функций, возвращающих значения

Все рекурсивные функции, с которыми мы работали до сих пор, имели тип void, но это вовсе не является обязательным требованием. Такая функция может возвращать значение любого типа.

Принципы разработки рекурсивных функций возвращающих значения, совершенно те же, что и для функций типа void.

Общая схема успешного определения рекурсивной функции, возвращающей значение следующая.

- При одном или нескольких условиях для вычисления значения функции требуются рекурсивные вызовы. Как и в случае функций типа void, аргументы каждого следующего вызова рекурсивной функции должны быть «меньше» аргументов предыдущих вызовов.
- При одном или нескольких условиях для вычисления значения функции не требуются рекурсивные вызовы. Эти условия называются *базовыми* или *условиями останова* (так же, как и для функций типа void).

Данная схема иллюстрируется следующим примером.

Пример: еще одна функция возведения в степень

В главе 3 описывалась стандартная функция pow, выполняющая возведение в степень. В частности, функция pow(2.0, 3.0) возвращает $2.0^{3.0}$, так что вызов

```
double x = pow(2.0, 3.0);
```

устанавливает значение переменной x равным 8.0.

Функция pow принимает два аргумента типа double и возвращает значение этого же типа. В листинге 13.3 приведено рекурсивное определение подобной функции, работающей с типом данных int. Эта новая функция названа power. Так, вызов

```
int y = power(2, 3);
```

устанавливает значение переменной y равным 8.

Мы определили функцию power как рекурсивную в демонстрационных целях, но в некоторых случаях она действительно удобнее функции pow. Функция pow возвращает значения типа double, то есть приближенные числа. А операция возведения целого числа в целую степень всегда дает точный результат, и если в программе требуется повышенная точность, лучше использовать функцию power.

Определение функции power основано на следующей формуле:

$$x^n = x^{n-1} * x$$

Листинг 13.3. Рекурсивная функция power

```
// Программа, демонстрирующая рекурсивную функцию power.
#include <iostream>
#include <cstdlib>
using namespace std;

int power(int x, int n);
// Предусловие: n >= 0.
// Возвращает x в степени n.

int main()
{
    for (int n = 0; n < 4; n++)
        cout << "3 to the power " << n
             << " is " << power(3, n) << endl;

    return 0;
}

// Используем библиотеки классов iostream и cstdlib.
int power(int x, int n)
{
    if (n < 0)
    {
        cout << "Illegal argument to power.\n";
        exit(1);
    }
    if (n > 0)
        return ( power(x, n - 1)*x );
    else   // n == 0
        return (1);
}
```

Пример диалога

```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

Перевод этой формулы на язык C++ означает, что значение, возвращенное вызовом `power(x, n)`, должно быть таким же, как значение выражения

```
power(x, n-1)*x
```

Определение функции `power`, приведенное в листинге 13.3, возвращает это значение при условии, что `n > 0`. Условием останова является равенство параметра `n` нулю. Если это так, `power(x, n)` возвращает `1` (поскольку x^0 равно 1).

Выясним, что происходит, когда функция `power` вызывается с некоторыми аргументами. Сначала рассмотрим такое выражение:

```
power(2, 0)
```

Когда функция вызывается с этими аргументами, значение `x` устанавливается равным 2, а значение `n` — равным 0, после чего выполняется код тела функции. Так как значение `n` допустимо, выполняется оператор `if...else`. А поскольку оно не больше нуля, тут же выполняется оператор `return`, следующий за ключевым словом `else`, так что вызов функции возвращает 1. Таким образом, следующий оператор:

```
int y = power(2, 0);
```

устанавливает значение переменной `y` равным 1.

Теперь рассмотрим пример, требующий рекурсивного вызова:

```
power(2, 1)
```

При выполнении вызова значение `x` устанавливается равным 2, а значение `n` — равным 1, после чего выполняется код тела функции. Поскольку значение `n` больше нуля, возвращаемое значение определяется следующим оператором `return`:

```
return (power(x, n-1)*x );
```

что в данном случае эквивалентно

```
return ( power(2, 0)*2 );
```

В этой точке вычисление значения `power(2, 1)` приостанавливается, копия остановленных вычислений помещается в стек, и компьютер приступает к выполнению нового вызова функции, вычисляющего значение `power(2, 0)`. Как мы только что видели, значением `power(2, 0)` является 1. Выполнив этот вызов, компьютер заменяет выражение `power(2, 0)` значением 1, после чего продолжает выполнение первого вызова функции `power`. В результате определяется окончательное значение `power(2, 1)`, и приведенный выше оператор `return` возвращает

```
power(2, 0)*2
```

что эквивалентно значению

```
1*2
```

то есть 2.

Таким образом, итоговым значением, возвращенным вызовом power(2, 1), является 2. Поэтому следующий оператор:

```
int z = power(2, 1);
```

установит значение переменной z равным 2.

Большие значения второго аргумента вызовут выполнение более длинных цепочек рекурсивных вызовов. В качестве примера рассмотрим оператор

```
cout << power(2, 3);
```

Значение power(2, 3) вычисляется следующим образом:

```
power(2, 3) равно power(2, 2)*2
  power(2, 2) равно power(2, 1)*2
    power(2, 1) равно power(2, 0)*2
      power(2, 0) равно 1 (условие останова)
```

Когда компьютер достигает условия останова, то есть выполняет вызов power(2, 0), у него имеется три приостановленных процесса вычислений. Вычислив значение, возвращаемое последним вызовом, в котором удовлетворяется условие останова, компьютер возобновляет последний приостановленный процесс вычислений, чтобы определить значение power(2, 1). И после этого завершает каждое из оставшихся вычислений, пока не закончит выполнение вычислений в исходном вызове, power(2, 3). Подробно полный процесс показан на рис. 13.1.

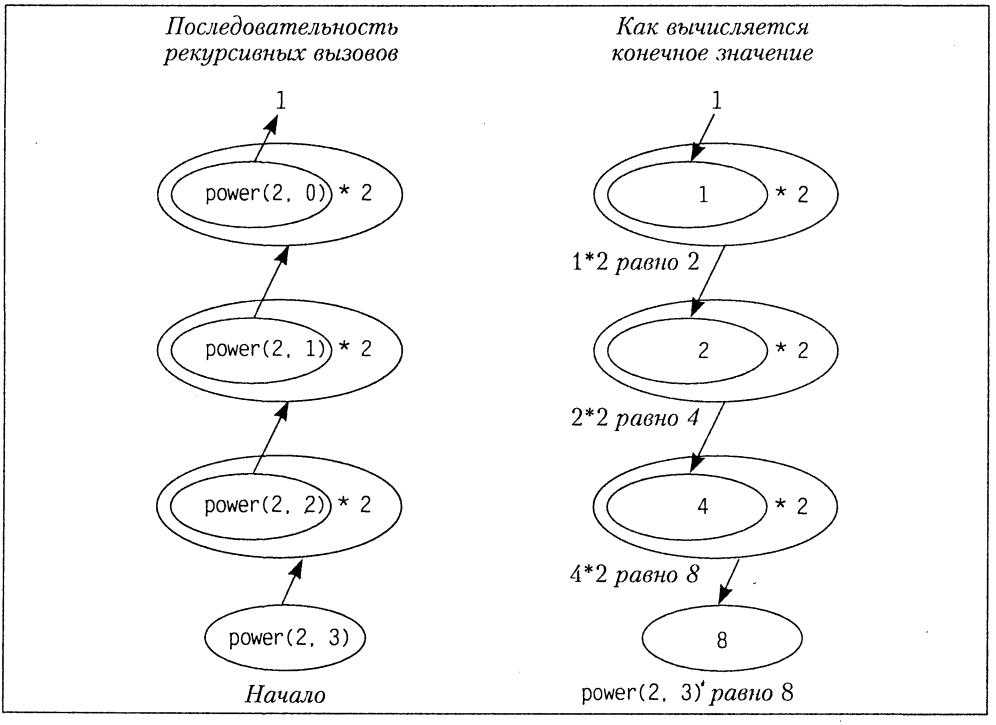


Рис. 13.1. Выполнение рекурсивного вызова power(2, 3)

Упражнения для самопроверки

12. Что выведет следующая программа:

```
#include <iostream>
using namespace std;
int mystery(int n);
// Предусловие: n >= 1.

int main()
{
    cout << mystery(3);
    return 0;
}

int mystery(int n)
{
    if (n <= 1)
        return 1;
    else
        return ( mystery(n - 1) + n );
}
```

13. Что выведет следующая программа:

```
#include <iostream>
using namespace std;
int rose(int n);
// Предусловие: n >= 0.

int main()
{
    cout << rose(4);
    return 0;
}

int rose(int n)
{
    if (n <= 0)
        return 1;
    else
        return ( rose(n - 1) * n );
}
```

Какую хорошо известную математическую функцию реализует функция rose?

14. Переопределите функцию power, чтобы она работала и с отрицательными степенями. Для этого нужно изменить тип возвращаемого ею значения: это должен быть тип double. Объявление и сопутствующий комментарий данной функции таковы:

```
double power(int x, int n);
// Предусловие: если n < 0, то x не равно 0.
// Возвращает x в степени n
```

Подсказка. $x^{-n} = 1/(x^n)$.

13.3. Рекурсивное мышление

Люди делятся на две категории: те, кто делит людей на две категории, и те, кто этого не делает.

Автор неизвестен

Рекурсивные технологии проектирования

При определении и использовании рекурсивных функций совсем не обязательно постоянно помнить о стеке и приостановленных вычислениях. Достоинство рекурсии заключается как раз в том, что программист может игнорировать детали и позволить компьютеру «все устроить наилучшим образом». Вернемся к примеру функции power, приведенной в листинге 13.3. Ее определение следует читать так:

`power(x, n)` возвращает `power(x, n-1)*x`

Поскольку x^n равно $x^{n-1}*x$, то это правильное возвращаемое значение при условии, что компьютер обязательно достигнет условия останова и без ошибок выполнит соответствующие ему вычисления. Поэтому после проверки правильности рекурсивной части определения функции следует только убедиться, что рекурсивные вызовы всегда ведут к условию останова и по его достижении функция возвращает верное значение.

Разрабатывая рекурсивную функцию, не нужно прослеживать всю последовательность возможных рекурсивных вызовов. Если функция возвращает значение, достаточно проверить следующее.

1. Рекурсия не может быть бесконечной. (Одни рекурсивные вызовы могут приводить к другим, но каждая такая цепочка обязательно должна приводить к условию останова.)
2. Для каждого условия останова функция возвращает правильное значение.
3. Если все рекурсивные вызовы возвращают правильные значения, то окончательное значение, возвращаемое функцией, тоже является правильным.

В качестве примера рассмотрим все ту же функцию power из листинга 13.3.

1. *Рекурсия не может быть бесконечной.* Второй аргумент функции `power(x,n)` в каждом следующем рекурсивном вызове уменьшается на 1, так что рано или поздно цепочка рекурсивных вызовов приводит к вызову `power(x, 0)`, являющемуся условием останова. Поэтому бесконечная рекурсия невозможна.
2. *Для каждого условия останова функция возвращает верное значение.* Единственное условие останова — вызов `power(x, 0)`, который всегда возвращает 1, а правильным значением x^0 является 1. Поэтому для условия останова функция возвращает правильное значение.
3. *Если все рекурсивные вызовы возвращают верные значения, то окончательное значение, возвращаемое функцией, тоже является верным.* Единственным условием, требующим рекурсии, является `n > 1`. Когда это так, вызов `power(x,n)` возвращает

```
power(x, n-1)*x
```

Чтобы убедиться в правильности этого значения, примите во внимание следующее: если power(x, n-1) возвращает правильное значение, то power(x, n-1) возвращает x^{n-1}, и тогда power(x,n) возвращает $x^{n-1}*x$, то есть x^n, и это является правильным значением функции power(x, n).

Вот и все, что нужно поверить, чтобы убедиться в правильности определения функции power. (Описанная логическая схема называется *математической индукцией*, и возможно, вы уже встречались с ней на уроках математики.)

Мы привели три критерия проверки правильности рекурсивной функции, возвращающей значение. Те же правила можно применить и к рекурсивной функции типа void. Если показать, что определение такой рекурсивной функции соответствует названным трем критериям, это будет доказательством ее правильной работы.

Пример: реализация двоичного поиска с помощью рекурсии

В данном примере разрабатывается рекурсивная функция, выполняющая поиск в массиве заданного значения. Например, массив может содержать список номеров кредитных карточек, которые больше не действительны. Служащий в магазине просматривает этот список, чтобы узнать, можно ли принять карточку клиента. В главе 10 (см. листинг 10.10) показан простой метод поиска значения в массиве путем последовательной проверки каждого его элемента. В данном разделе используется гораздо более быстрый метод поиска в отсортированном массиве.

Индексами в массиве a являются целые числа от 0 до final_index. Для облегчения задачи поиска мы будем полагать, что массив уже отсортирован. Это означает, что нам известно следующее:

```
a[0] <= a[1] <= a[2] <= ... <= a[final_index]
```

При выполнении такого поиска обычно требуется узнать, присутствует ли в массиве заданный элемент, и если да, получить его индекс. В нашем случае при поиске номера кредитной карточки ее индекс в массиве может служить номером записи, содержащей дополнительную информацию о карточке. В другом массиве с такими же индексами могут содержаться номера телефонов или другая информация, связанная с подозрительной карточкой. Поэтому если искомое значение обнаружено в массиве, функция должна указать, где именно оно находится.

Постановка задачи

Мы будем разрабатывать функцию с двумя передаваемыми по ссылке параметрами, возвращающую результат поиска. Один параметр, found, будет иметь тип bool. Если функция нашла заданное значение, этот параметр принимает значение true. Второму параметру, location, в таком случае присваивается индекс найденного значения. Обозначив искомое значение как key, задачу функции можно сформулировать так:

```
Предусловие: значения от a[0] до a[final_index]
             отсортированы по возрастанию.
```

```
Постусловие: если key не равно одному из значений
             от a[0] до a[final_index],
                то found == false;
             в противном случае
                a[location] == key и found == true.
```

Разработка алгоритма

Теперь займемся созданием алгоритма для решения этой задачи. Она должна быть сформулирована в предельно конкретных выражениях. Предположим, список номеров карточек настолько длинный, что в бумажном виде занимает целую книгу. Именно в таком виде перечень номеров недействительных кредитных карточек хранится в магазинах, где нет компьютеров. Клерк, получивший от клиента кредитную карточку, должен проверить, нет ли ее в «черном списке». Как он будет это делать? Откроет книгу посередине и посмотрит, нет ли номера на текущей странице. Если его нет и он меньше номеров на этой странице, нужно будет поискать его в первой половине книги, а если он больше — во второй половине. Идея проста, и можно выразить ее вот в таком черновом наброске алгоритма:

```
found = false;
mid = примерно середина между 0 и final_index;
if (key == a[mid])
{
    found = true;
    location = mid;
}
else if (key < a[mid])
    искать в части массива от a[0] до a[mid-1];
else if (key > a[mid])
    искать в части массива от a[mid+1] до a[final_index];
```

Поскольку поиск в более коротком списке, безусловно, является меньшей версией всей задачи, наш алгоритм явно имеет рекурсивную природу. Поиск в меньшем списке можно выполнять с помощью рекурсивного вызова той же функции.

Наш псевдокод недостаточно точен для перевода на C++. Главная проблема связана с рекурсивными вызовами. Здесь их два:

```
search a[0] through a[mid-1];
```

и

```
search a[mid+1] through a[final_index];
```

Для реализации этих вызовов необходимы еще два параметра. В рекурсивном вызове задается часть массива, где следует искать заданное значение. В одном случае это элементы с индексами от 0 до mid-1, а во втором — элементы с индексами от mid+1 до final_index. Два дополнительных параметра будут определять первый и последний индексы просматриваемого диапазона, поэтому мы назовем их first и last. Используя эти идентификаторы для наименьшего и наибольшего индексов вместо 0 и final_index, можно записать псевдокод более точно:

```
found = false;
mid = примерно середина между first и last;
if (key == a[mid])
{
```

```
    found = true;
    location = mid;
}
else if (key < a[mid])
    искать в части массива от a[first] до a[mid-1];
else if (key > a[mid])
    искать в части массива от a[mid+1] до a[last];
```

Для поиска в целом массиве этот алгоритм нужно выполнить со значением 0 параметра first и значением переменной final_index параметра last. В рекурсивных вызовах будут использоваться другие значения first и last. Например, в первом вызове параметр first будет равен 0, а параметр last будет равен mid-1.

Как и для любого другого рекурсивного алгоритма, следует убедиться, что наш алгоритм не содержит возможности бесконечной рекурсии. Если искомый номер найден в списке, дальнейшие рекурсивные вызовы не выполняются и процесс завершается, но нам требуется условие, определяющее, что номер отсутствует в списке. В каждом рекурсивном вызове либо увеличивается значение first, либо уменьшается значение last. И если они сойдутся или даже поменяются местами, то есть first станет больше last, будет ясно, что больше проверять нечего и значение key в массиве отсутствует. Если добавить в псевдокод эту проверку, получится решение, показанное в листинге 13.4.

Листинг 13.4. Псевдокод для двоичного поиска

```
int a[Some_Size_Value];

Алгоритм для поиска значения в части массива от a[first] до a[last].

// Предусловие:
// a[first] <= a[first+1] <= a[first+2] <= ... <= a[last]

Для поиска значения key:

    if (first > last)   // Условие останова.
        found = false;
    else
    {
        mid = примерно середина между first и last;
        if (key == a[mid])   // Условие останова.
        {
            found = true;
            location = mid;
        }
        else if key < a[mid]   // Условие, требующее рекурсии:
            искать в части массива от a[first] до a[mid-1].
        else if key > a[mid]   // Условие, требующее рекурсии:
            искать в части массива от a[mid+1] до a[last].
    }
```

Написание программного кода

Теперь осталось лишь механически перевести псевдокод на язык C++. Результирующий код приведен в листинге 13.5. Функция search реализует рекурсивный алгоритм из листинга 13.4. Диаграмма ее выполнения на примере демонстрационного массива показана на рис. 13.2.

Обратите внимание, что функция search решает более общую задачу, чем та, которая перед нами поставлена. Наша цель заключалась в разработке функции для поиска значения во всем массиве. Однако полученная функция может искать значения в любом подмассиве – достаточно задать его начальный и конечный индексы в параметрах first и last. Это типичная ситуация для рекурсивных функций – чтобы сформулировать рекурсивный алгоритм, часто приходится решить более общую задачу. В данном примере нам требовалось решение для случая, когда first и last равны 0 и final_index соответственно. Однако в рекурсивных вызовах этим параметрам могут быть присвоены и другие значения.

Листинг 13.5. Рекурсивная функция для двоичного поиска

```
// Программа, использующая рекурсивную функцию для двоичного поиска.
#include <iostream>
using namespace std;
const int ARRAY_SIZE = 10;

void search(const int a[], int first, int last,
                  int key, bool& found, int& location);
// Предусловие: элементы от a[first] до a[last] отсортированы по возрастанию.
// Постусловие: если значение переменной key не равно одному из элементов
// от a[0] до a[final_index], то found = false;
// в противном случае a[location] = key и found = true.

int main()
{
    int a[ARRAY_SIZE];
    const int final_index = ARRAY_SIZE - 1;

    ... Эта часть программы содержит некоторый код, чтобы заполнять
        и сортировать массив a. Для данного примера он не имеет значения. ...

    int key, location;
    bool found;
    cout << "Enter number to be located: ";
    cin >> key;
    search(a, 0, final_index, key, found, location);

    if (found)
        cout << key << " is in index location "
             << location << endl;
    else
        cout << key << " is not in the array." << endl;

    return 0;
}

void search(const int a[], int first, int last,
                          int key, bool& found, int& location)
{
    int mid;
    if (first > last)
    {
        found = false;
    }
```

продолжение

Листинг 13.5 (*продолжение*)

```
    else
    {
        mid = (first+last)/2;

        if (key == a[mid])
        {
            found = true;
            location = mid;
        }
        else if (key < a[mid])
        {
            search(a, first, mid-1, key, found, location);
        }
        else if (key > a[mid])
        {
            search(a, mid+1, last, key, found, location);
        }
    }
}
```

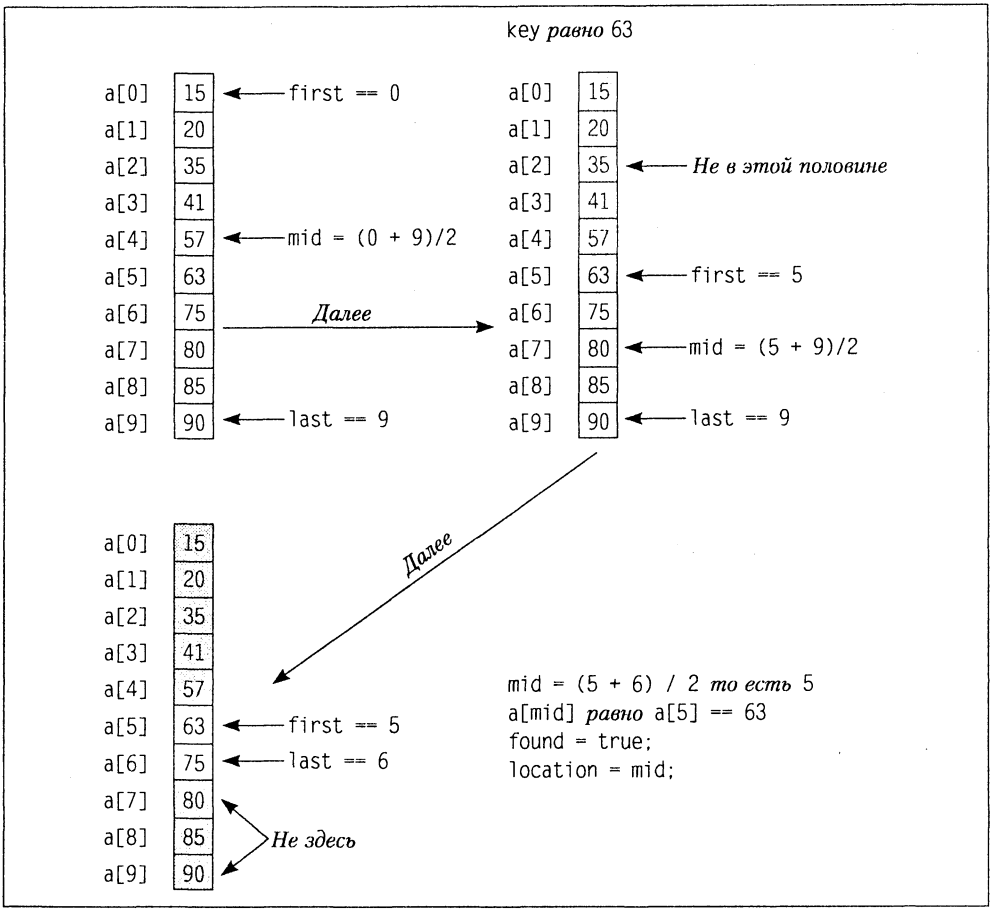


Рис. 13.2. Выполнение функции search

Проверка рекурсии

В подразделе «Рекурсивные технологии проектирования» приведены три критерия, позволяющие убедиться в правильности определения рекурсивной функции типа void. Давайте проверим их для функции search, приведенной в листинге 13.5.

1. *Бесконечная рекурсия невозможна.* В каждом рекурсивном вызове увеличивается значение параметра first или уменьшается значение параметра last. Если цепочка рекурсивных вызовов не завершится иным способом, рано или поздно функция будет вызвана для значений first > last, что является условием останова.
2. *Для каждого условия останова функция выполняет корректное действие.* У нашей функции два условия останова: first > last и key == a[mid]. Рассмотрим каждый случай.

   Если first > last, то между элементами a[first] и a[last] нет ни одного элемента массива, и поэтому значения key в этом сегменте массива быть не может. (Собственно, такого сегмента вообще нет!) Поэтому, если first > last, функция search устанавливает параметр found равным значению false.

   Если key == a[mid], функция search правильно устанавливает параметр found равным true, и параметр location равным mid. Следовательно, для обоих условий останова функция выполняет корректные действия.
3. *Для каждого из условий, требующих рекурсии: если все рекурсивные вызовы выполняют правильные действия, то это условие обрабатывается правильно.* Рекурсивные вызовы выполняются в двух случаях: когда key < a[mid], и когда key > a[mid]. Проверим каждый из них.

   Сначала предположим, что key < a[mid]. В этом случае, поскольку массив отсортирован, нам известно, что key может быть только одним из элементов от a[first] до a[mid-1]. Поэтому функции нужно выполнить поиск лишь среди этих элементов, то есть выполнить рекурсивный вызов

   ```
   search(a, first, mid-1, key, found, location);
   ```

   А раз этот рекурсивный вызов верен, то и все действие правильно.

   Теперь предположим, что key > a[mid]. В этом случае, поскольку массив отсортирован, нам известно, что key может быть только одним из элементов от a[mid+1] до a[last]. Поэтому функции нужно выполнить поиск лишь среди этих элементов, то есть выполнить рекурсивный вызов

   ```
   search(a, mid + 1, last, key, found, location);
   ```

   А раз этот рекурсивный вызов верен, все действие правильно. Таким образом, в обоих случаях функция выполняет правильные действия (если, конечно, сами рекурсивные вызовы выполняют корректные действия).

Функция search успешно прошла все три проверки, следовательно, мы разработали хорошее определение.

Эффективность

По сравнению с алгоритмом последовательного перебора элементов массива алгоритм двоичного поиска работает намного быстрее. С самого начала он устраняет

необходимость просмотра половины массива, затем четверти, затем одной восьмой и т. д. — не удивительно, что он работает исключительно эффективно. Для массива из 100 элементов алгоритм двоичного поиска сравнивает с искомым значением не более 7 элементов, тогда как последовательный поиск потребует сравнения около 50 элементов. Заметьте, чем больше массив, тем значительнее будет ускорение поиска за счет более эффективного алгоритма. Для массива в 1000 элементов двоичный поиск потребует сравнения не более 10 элементов, что несопоставимо со средним количеством в 500 элементов для обычного последовательного поиска.

Итеративная версия функции search приведена в листинге 13.6. В некоторых системах она будет работать быстрее рекурсивной. Однако действие обеих версий одинаково. В итеративной версии локальные переменные first и last выполняют роль одноименных параметров рекурсивной функции. Как видно из данного примера, даже если вы планируете использовать итеративный алгоритм, иногда стоит сначала разработать его рекурсивную версию, которую затем легко преобразовать в итеративную.

Листинг. 13.6. Итеративная версия двоичного поиска

Объявление функции

```
void search(const int a[], int low_end, int high_end,
                           int key, bool& found, int& location);
// Предусловие: значения от a[low_end] до a[high_end] отсортированы по возрастанию.
// Постусловие: если key не равно одному из значений
// от a[0] до a[high_end], то found == false;
// в противном случае a[location] == key и found == true.
```

Определение функции

```
void search(const int a[], int low_end, int high_end,
                           int key, bool& found, int& location)
{
    int first = low_end;
    int last = high_end;
    int mid;

    found = false;
    while ((first <= last) && !(found))
    {
        mid = (first + last)/2;
        if (key == a[mid])
        {
            found = true;
            location = mid;
        }
        else if (key < a[mid])
        {
            last = mid - 1;
        }
        else if (key > a[mid])
        {
            first = mid + 1;
        }
    }
}
```

Пример: рекурсивная функция-член

Функция-член класса может быть рекурсивной. Рекурсия в функциях-членах применяется совершенно так же, как в обычных функциях. Листинг 13.7 содержит пример этой функции. Используемый в программе класс BankAccount — такой же класс BankAccount, как в листинге 6.5, с той разницей, что у него имеется перегруженная функция-член update. Первая версия данной функции не имеет аргументов и добавляет к балансу на банковском счету процентные начисления за один год. Вторая (новая) версия функции принимает аргумент типа int, в котором задается количество лет. Она обновляет баланс, добавляя к нему процентные начисления за указанное количество лет. Новая версия функции update рекурсивна. Ее параметр называется years, а алгоритм работы таков:

```
Если значение years равно 1, то   // Условие останова:
     вызвать другую функцию update (без аргументов).
Если значение years больше 1, то   // Условие рекурсии:
     выполнить рекурсивный вызов для добавления начислений
     за years-1 лет и затем вызвать другую функцию update
     (без аргументов) для добавления начислений еще за один год.
```

Давайте убедимся, что этот алгоритм выполняет свою задачу, и проверим три условия, указанные в разделе «Рекурсивные технологии проектирования».

1. *Бесконечная рекурсия невозможна.* В каждом рекурсивном вызове количество лет, years, сокращается на единицу, пока не станет равным 1, что является условием останова.
2. *Для каждого условия останова функция выполняет правильное действие.* Единственным условием останова является условие years == 1. В этом случае выполняется правильное действие, поскольку просто вызывается другая перегруженная функция-член update, а ее правильность мы проверили в главе 6.
3. *Для каждого из условий, требующих рекурсии: если все рекурсивные вызовы выполняют правильные действия, то это условие обрабатывается правильно.* В случае, когда требуется рекурсия, то есть years > 1, функция работает верно. Если рекурсивный вызов правильно добавляет начисления за years-1 лет, то после этого нужно только добавить начисления еще за один год, что и делает вызов перегруженной версии функции-члена update без аргументов. Таким образом, если рекурсивный вызов выполняет правильное действие, то вся операция, выполняемая функцией в случае years > 1, правильна.

Листинг 13.7. Рекурсивная функция-член

```
// Программа, использующая рекурсивную функцию-член update(years).

#include <iostream>
using namespace std;

// Класс BankAccount в этой программе является усовершенствованной
// версией одноименного класса, приведенного в листинге 6.6.
```

продолжение

Листинг 13.7 (*продолжение*)

```
// Класс для банковского счета:
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    // Инициализирует баланс счета значением $dollars.cents
    // и процентную ставку счета значением rate процентов.

    BankAccount(int dollars, double rate);
    // Инициализирует баланс счета значением $dollars.00
    // и процентную ставку счета значением rate процентов.

    BankAccount();
    // Инициализирует баланс счета значением $0.00
    // и процентную ставку счета значением 0.0%.

    // Две разные функции с одним именем.

    void update();
    // Постусловие: к балансу счета прибавлены процентные
    // начисления за один год.

    void update(int years);
    // Постусловие: к балансу счета прибавлены процентные
    // начисления за заданное количество лет. Начисления
    // выполняются ежегодно.

    double get_balance();
    // Возвращает значение текущего баланса счета.

    double get_rate();
    // Возвращает значение текущей процентной ставки.

    void output(ostream& outs);
    // Предусловие: если outs – выходной файловый поток,
    // он уже соединен с файлом.
    // Постусловие: значения баланса счета и процентной ставки
    // записаны в поток outs.
private:
    double balance;
    double interest_rate;
    double fraction(double percent);
    // Преобразует проценты в дробь.
};

int main()
{
    BankAccount your_account(100, 5);
    your_account.update(10);
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "If you deposit $100.00 at 5% interest, then\n"
```

```
        << "in ten years your account will be worth $"
        << your_account.get_balance() << endl;
    return 0;
}

void BankAccount::update()
{
    balance = balance + fraction(interest_rate)*balance;
}

void BankAccount::update(int years)
{
    if (years == 1)
    {
        update();          // Перегрузка функции.
    }
    else if (years > 1)
    {
        update(years - 1);  // Рекурсивный вызов функции.
        update();   // Перегрузка функции.
    }
}
```

... Определения остальных функций-членов приведены в листингах 6.4 и 6.5, но для понимания этого примера они не обязательны. ...

Пример диалога

```
If you deposit $100.00 at 5% interest, then
in ten years your account will be worth $162.89
```

В данном примере у нас есть две разные версии перегруженной функции-члена update. Одна из них не принимает ни одного аргумента, а вторая принимает один. Вызовы этих функций следует различать. Для компилятора они являются двумя разными функциями с одинаковыми именами. И когда определение функции update с одним аргументом содержит вызов версии update без аргументов — это не рекурсивный вызов. Рекурсивным является только вызов версии update с точно таким же объявлением. Все это вы лучше поймете, если учтете, что функцию update без аргументов можно было бы назвать как-нибудь иначе, например post_one_year(), и тогда определение рекурсивной версии update было бы таким:

```
void BankAccount::update(int years)
{
    if (years == 1)
    {
        post_one_year();
    }
    else if (years > 1)
    {
        update(years - 1);
        post_one_year();
    }
}
```

Рекурсия и перегрузка

Следует различать рекурсию и перегрузку функций. При перегрузке функции двум разным функциям назначается одно и то же имя. Если определение одной из них содержит вызов другой, это не рекурсия. Если же функция рекурсивна, ее определение содержит вызов ее самой с тем же самым определением, а не просто вызов другой функции с тем же именем. Язык C++ допускает и перегрузку, и рекурсию, а если вы спутали два названных понятия, это еще не значит, что в программе имеется ошибка. Однако во избежание разночтений лучше все же употреблять правильную терминологию, в особенности в беседах с другими программистами. Ну а то, что верное понимание используемых технологий облегчает программирование и помогает применять их более эффективно, даже не подлежит обсуждению.

Упражнения для самопроверки

15. Напишите рекурсивное определение для следующей функции:

```
int squares(int n);
// Предусловие: n >= 1
// возвращает значение суммы квадратов чисел от 1 до n.
```

Например, `squares(3)` возвращает 14, поскольку $1^2 + 2^2 + 3^2$ равно 14.

16. Напишите итеративную версию функции-члена `BankAccount::update(int years)` с одним аргументом, приведенной в листинге 13.7.

Резюме

- Если задачу можно свести к упрощенным версиям этой же задачи, то для нее возможно рекурсивное решение.
- Рекурсивный алгоритм задания функции обычно определяет действия в двух случаях: одно или несколько условий требуют выполнения хотя бы одного рекурсивного вызова и одно или несколько условий останова позволяют решить задачу без рекурсивных вызовов.
- При написании определения рекурсивной функции всегда внимательно анализируйте его на предмет бесконечной рекурсии.
- Определив рекурсивную функцию, проверяйте ее правильность с помощью трех критериев, перечисленных в разделе «Рекурсивные технологии проектирования».
- При разработке рекурсивной функции для решения какой-либо задачи вам часто придется решать более общую задачу. Это необходимо, чтобы правильно выполнять рекурсивные вызовы, поскольку уменьшенные задачи могут не быть точной копией исходной. Например, в случае двоичного поиска может быть поставлена задача поиска в целом массиве, а рекурсивным решением оказывается алгоритм поиска в любой части массива (которой, в частности, может быть и весь массив).

Ответы к упражнениям для самопроверки

1. Hip Hip Hurray

2. ```
   void stars(int n)
   {
       cout << '*';
       if (n > 1)
           stars(n - 1);
   }
   ```

   Правильным является и следующее более сложное определение:

   ```
   void stars(int n)
   {
       if (n <= 1)
       {
           cout << '*';
       }
       else
       {
           stars(n - 1);
           cout << '*';
       }
   }
   ```

3. ```
   void backward(int n)
   {
       if (n < 10)
       {
           cout << n;
       }
       else
       {
           cout << (n%10);   // Вывод последней цифры.
           backward(n/10);   // Вывод остальных цифр в обратном порядке.
       }
   }
   ```

4. ```
   #include <iostream>
   using namespace std;
   void write_up(int n)
   {
       if (n >= 1)
       {
           write_up(n - 1);
           cout << n << " ";
       }
   }
   int main()
   {
       cout << "calling write_up(" << 10 << ")\n";
       write_up(10);
       cout << endl;
       return 0;
   }
   ```
   ```

Результаты тестирования функции write_up(10):

```
1 2 3 4 5 6 7 8 9 10
```

5.
```
#include <iostream>
using namespace std;
void write_down(int n)
{
    if (n >= 1)
    {
        cout << n << " ";
        write_down(n - 1);
    }
}

int main()
{
    cout << "calling write_down(" << 10 << ")\n";
    write_down(10);
    cout << endl;
    return 0;
}
```

Результаты тестирования функции write_down(10):

```
10 9 8 7 6 5 4 3 2 1
```

6. Сообщение о переполнении стека означает, что компьютер попытался поместить в стек очередную запись активации, но для нее не хватило места. Вероятной причиной этой ошибки является бесконечная рекурсия.

7.
```
void cheers(int n)
{
    while (n > 1)
    {
        cout << "Hip ";
        n--;
    }
    cout << "Hurray\n";
}
```

8.
```
void stars(int n)
{
        for (int count = 1; count <= n; count++)
            cout << '*';
}
```

9.
```
void backward(int n)
{
    while (n >= 10)
    {
        cout << (n%10);  // Вывод последней цифры.
        n = n/10;        // Удаление последней цифры.
    }
    cout << n;
}
```

10. Процесс выполнения функции из упражнения 4: если n = 3, выполняется код:

```
if (3 >= 1)
{
    write_up(3 - 1);
    cout << 3 << " ";
}
```

В следующем рекурсивном вызове n = 2, и выполняется код:

```
if (2 >= 1)
{
    write_up(2 - 1);
    cout << 2 << " ";
}
```

В следующем рекурсивном вызове n = 1, и выполняется код:

```
if (1 >= 1)
{
    write_up(1 - 1);
    cout << 1 << " ";
}
```

Наконец, в последнем рекурсивном вызове n = 0, и выполняется код:

```
if (0 >= 1)   // Условие возвращает false, код пропускается.
{
    // Код пропускается.
}
```

На этом рекурсивные вызовы прекращаются, а те, что уже начаты, по очереди выполняются до конца. В результате на экран выводится следующее: 1 2 3.

11. Процесс выполнения функции из упражнения 5: если n = 3, выполняется код:

```
if (3 >= 1)
{
    cout << 3 << " ";
    write_down(3 - 1);
}
```

В следующем рекурсивном вызове n = 2, и выполняется код:

```
if (2 >= 1)
{
    cout << 2 << " ";
    write_down(2 - 1)
}
```

В следующем рекурсивном вызове n = 1, и выполняется код:

```
if (1 >= 1)
{
    cout << 1 << " ";
    write_down(1 - 1)
}
```

Наконец, в последнем рекурсивном вызове n = 0, и код, соответствующий истинности условия оператора if, не выполняется:

```
if ( 0 >= 1 )   // Условие возвращает false.
{
    // Этот код пропускается.
}
```

12. 6

13. Программа выводит число 24. Эта функция вычисляет факториал, обозначаемый $n!$ и определяемый следующим образом:

$$n! = n*(n - 1)*(n - 2)*...*1$$

14.
```
// Используем библиотеки классов iostream и cstdlib.
double power(int x, int n)
{
    if (n < 0 && x == 0)
    {
        cout << "Illegal argument to power.\n";
        exit(1);
    }
    if (n < 0)
        return (1/power(x, -n));
    else if (n > 0)
        return (power(x, n - 1)*x);
    else   // n == 0
        return (1.0);
}
```

15.
```
int squares(int n)
{
    if (n <= 1)
        return 1;
    else
        return (squares(n - 1) + n*n);
}
```

16.
```
void BankAccount::update(int years)
{
    for (int count = 1; count <= years; count++)
        update();
}
```

Практические задания

1. Напишите рекурсивное определение функции с одним параметром `n` типа `int`, возвращающей n-е число Фибоначчи. Определение чисел Фибоначчи приведено в описании практического задания 8 главы 7. Включите функцию в отладочную программу и протестируйте.
2. Напишите рекурсивную версию функции `index_of_smallest`, использовавшейся в программе сортировки в листинге 10.10 главы 10. Включите функцию в отладочную программу и протестируйте.
3. Формула для вычисления количества способов выбора r разных элементов из множества, состоящего из n элементов, такова:

   $$C(n, r) = n!/(r!*(n - r)!)$$

   Функция, возвращающая факториал, $n!$, определяется следующим образом:

   $$n! = n*(n - 1)*(n - 2)*...*1$$

Напишите рекурсивную версию этой формулы и рекурсивную функцию, вычисляющую ее значение. Включите функцию в отладочную программу и протестируйте.

4. Напишите рекурсивную функцию, первым аргументом которой является массив символов, а еще двумя — граничные значения диапазона индексов в данном массиве. Функция должна заменить порядок входящих в этот диапазон элементов обратным. Например, для следующего массива:

```
a[1] == 'A'  a[2] == 'B'  a[3] == 'C'  a[4] == 'D'  a[5] == 'E'
```

и границ 2 и 5, функция должна так переупорядочить элементы, чтобы после ее вызова содержимое массива было таким:

```
a[1] == 'A'  a[2] == 'E'  a[3] == 'D'  a[4] == 'C'  a[5] == 'B'
```

Включите функцию в отладочную программу и протестируйте ее. Когда функция будет полностью отлажена, определите еще одну функцию, принимающую единственный аргумент (массив, содержащий строковое значение) и заменяющую порядок символов в этом массиве обратным. Функция будет содержать вызов рекурсивного определения, разработанного вами в первой части задания. Включите вторую функцию в отладочную программу и протестируйте.

5. Напишите итеративную версию рекурсивной функции из практического задания 4. Включите ее в отладочную программу и протестируйте.

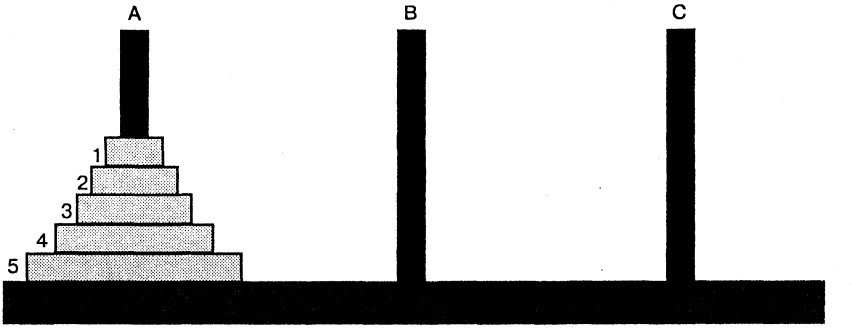
6. Напишите рекурсивную функцию для сортировки массива целых чисел по возрастанию. В основу алгоритма сортировки положите следующую идею: наименьший элемент помещается в первую позицию, после чего остальная часть массива сортируется с помощью рекурсивного вызова. Это рекурсивная версия сортировки методом выбора, описанная в главе 10.

Примечание. Недостаточно просто взять программу из главы 10 и включить в нее рекурсивную версию функции `index_of_smallest` — функция сортировки сама должна быть рекурсивной.

7. Ханойские башни. Существует буддистская легенда об обезьянах, играющих в эту игру-головоломку с 64 каменными дисками. В легенде рассказывается, что когда обезьяны закончат перемещать диски из первого столбика во второй через третий столбик, окончится время. Оставив рассуждения о конечности времени и теологии тем, кто в этом разбирается, мы займемся рекурсивным решением задачи.

Уменьшаемая стопка из n дисков составляет один из трех столбиков. Задача заключается в том, чтобы переместить диски по одному из первого столбика во второй. Любой диск может быть перемещен в любой из столбиков при условии, что больший диск никогда не кладется на меньший. Третий столбик используется как промежуточный, поскольку без него задачу решить невозможно. Вам нужно написать рекурсивную функцию, описывающую последовательность решения данной задачи. Поскольку мы не работаем с графикой, последовательность инструкций должна выводиться в текстовом виде.

Подсказка. Если каким-то образом переместить n-1 дисков из первого столбика в третий с использованием второго в качестве вспомогательного, последний диск можно будет переложить из первого столбика во второй. После этого с помощью той же схемы (какой бы она ни была), можно переместить n-1 дисков из третьего столбика во второй, используя первый диск как вспомогательный. Тогда задача будет решена. Остается только понять, в чем заключаются нерекурсивное и рекурсивное условия, а также выяснить, когда выводить инструкции для перемещения дисков.



Глава 14 Шаблоны

Все люди смертны.
Аристотель — человек.
Следовательно, Аристотель смертен.

Все X равны Y.
Z равно X.
Следовательно, Z равно Y.

Все коты шкодливые.
Гарфильд — кот.
Следовательно, Гарфильд шкодливый.

Краткий урок силлогизмов

В этой главе рассматриваются шаблоны языка C++. Они позволяют определять функции и классы с параметрами для имен типов, благодаря чему одна и та же функция может использоваться с аргументами разных типов. Кроме того, применение шаблонов дает возможность определять более универсальные классы, чем те, которые мы использовали до сих пор.

14.1. Шаблоны для абстрактных алгоритмов

Для многих разрабатывавшихся в этой книге функций мы сначала определяли более или менее универсальный алгоритм, а затем воплощали в функции его специализированную версию. В качестве примера рассмотрим функцию `swap_values`, описанную в главе 4. Ее определение следующее:

```
void swap_values(int& variable1, int& variable2)
{
    int temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Она работает только с переменными типа int, но алгоритм, на основе которого написано ее тело, с тем же успехом менял бы местами значения переменных типа char. Поэтому, если бы мы решили использовать эту функцию для переменных типа char, то могли бы перегрузить ее с помощью определения

```
void swap_values(char& variable1, char& variable2)
{
    char temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Однако создание этих двух определений функции swap_values вряд ли является эффективным решением проблемы, поскольку они практически идентичны. Единственное отличие состоит в том, что в первой из них в трех выделенных местах используется тип данных int, а во второй – char. Если бы мы захотели применить названную функцию также к переменным типа double и продолжали действовать тем же способом, нам пришлось бы писать третье определение, практически идентичное двум предыдущим. Результатом этой скучной работы было бы создание большого количества почти одинакового кода.

Хорошо было бы иметь возможность указать C++, что следующее определение функции:

```
void swap_values(тип_переменной& variable1,
                 тип_переменной& variable2)
{
    тип_переменной temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

применимо к переменным любого типа. Как вы увидите далее, нечто подобное действительно возможно. C++ позволяет создать определение функции, применимой ко всем типам переменных, хотя его синтаксис будет несколько отличаться от показанного выше.

Шаблоны функций

В листинге 14.1 приведен шаблон C++ для функции swap_values, который позволяет менять местами значения двух переменных любого типа, если только их тип одинаков. Определение и объявление функции начинаются так:

```
template<class T>
```

Данное выражение часто называют *префиксом шаблона*. Оно сообщает компилятору, что следующее определение функции – *шаблон*, а T – *параметр типа*. (В этом контексте class означает *тип*[1].) Как вы увидите далее, параметр типа T можно

[1] В стандарте ANSI сказано, что вместо слова class в префиксе шаблона может использоваться слово typename. И хотя мы согласны, что оно больше подходит по смыслу, использование слова class уже стало традиционным, так что ради согласованности с другими программистами и авторами мы придерживаемся этой традиции.

заменить любым типом, будь то класс или один из базовых типов языка C++. В теле определения функции параметр T используется так же, как любой другой тип.

Шаблон определения функции представляет собой большой набор определений сходных функций. Например, шаблон функции swap_values представляет набор определений функций для всех возможных имен типов данных. Следующее определение функции получено путем замены идентификатора T именем типа double:

```
void swap_values(double& variable1, double& variable2)
{
    double temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Еще два определения названной функции можно получить, если заменить параметр T именами типов int и char. Один шаблон, приведенный в листинге 14.1, перегружает имя функции swap_values так, что для каждого из существующих типов данных получается собственное определение этой функции.

Лисинг 14.1. Шаблон функции

```
// Программа, демонстрирующая шаблон функции.
#include <iostream>
using namespace std;

// Меняет местами значения переменных variable1 и variable2.
template<class T>
void swap_values(T& variable1, T& variable2)
{
      T temp;

      temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}

int main()
{
    int integer1 = 1, integer2 = 2;
    cout << "Original integer values are "
         << integer1 << " " << integer2 << endl;
    swap_values(integer1, integer2);
    cout << "Swapped integer values are "
         << integer1 << " " << integer2 << endl;

    char symbol1 = 'A', symbol2 = 'B';
    cout << "Original character values are "
         << symbol1 << " " << symbol2 << endl;
    swap_values(symbol1, symbol2);
    cout << "Swapped character values are "
         << symbol1 << " " << symbol2 << endl;

    return 0;
}
```

Вывод

```
Original integer values are 1 2
Swapped integer values are 2 1
Original character values are A B
Swapped character values are B A
```

На самом деле компилятор не генерирует определения функции swap_values для всех возможных типов данных, однако ведет он себя так, словно эти определения существуют. Определение функции генерируется для каждого типа, с которым в программе используется данный шаблон, но не для всех возможных типов. Причем для каждого типа генерируется только одно определение функции независимо от того, сколько раз применяется для него этот шаблон. Обратите внимание, что функция swap_values в листинге 14.1 вызывается дважды: один раз для аргументов типа int и второй раз для аргументов типа char.

Рассмотрим следующий вызов из программы листинга 14.1:

```
swap_values(integer1, integer2);
```

Когда компилятор C++ встречает этот вызов, он «смотрит» на типы аргументов (в данном случае int) и на основе шаблона генерирует определение функции, заменив параметр типа T именем типа int. Подобным же образом, если компилятор встречает вызов

```
swap_values(symbol1, symbol2);
```

он «смотрит» на типы аргументов (в данном случае char) и на основе шаблона генерирует определение функции, заменив параметр типа T именем типа char.

Заметьте, что вызов функции, определенной с помощью шаблона, не содержит ничего необычного — такая функция вызывается так же, как любая другая. А всю работу по формированию ее определения на основе шаблона выполняет компилятор.

Обратите также внимание, что в программе листинга 14.1 определение шаблона функции помещено перед функцией main — в то место, где обычно располагаются объявления функций. Однако это не обязательно, объявление и определение шаблона функции можно поместить туда, где находятся объявления и определения обычных функций. Но учтите, что некоторые компиляторы не поддерживают объявлений шаблонов функций, как не поддерживают и отдельной компиляции шаблонов функции. Если же объявления и отдельная компиляция шаблонов поддерживаются, они могут отличаться деталями реализации, так что в результате возникает путаница. В случае, когда вы планируете в дальнейшем переносить программу с одной платформы на другую, лучше вообще не используйте объявления шаблонов функций, а их определения помещайте в тот же файл, где они используются (то есть в файл, в котором вызывается данная функция). Причем сделать это можно не явно, а с помощью директивы #include, чтобы не повторять одно и то же определение во множестве файлов. Определение шаблона функции пишется один раз, и его файл с помощью названной директивы включается во все файлы, в которых применяется данная функция. Это наиболее эффективная и надежная стратегия.

Хотя в этой книге объявления шаблонов функций использоваться не будут, для тех читателей, чьи компиляторы их поддерживают, мы подробно опишем эти объявления и приведем примеры.

В рассмотренном выше шаблоне функции в качестве идентификатора параметра типа использовалась буква T. Это просто традиция, а вовсе не обязательное требование языка C++. На самом деле вместо данной буквы может использоваться любой идентификатор, не являющийся ключевым словом языка C++, причем иногда удобнее воспользоваться как раз другим идентификатором. Например, шаблон функции swap_values, приведенный в листинге 14.1, эквивалентен следующему шаблону:

```
template<class VariableType>
void swap_values(VariableType& variable1,
                                VariableType& variable2)
{
    VariableType temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Шаблон функции может иметь более одного параметра типа. Например, шаблон функции с двумя параметрами, T1 и T2, будет начинаться так:

```
template<class T1, class T2>
```

Однако для большинства шаблонов достаточно одного параметра типа. Учтите, что неиспользуемых параметров типа в шаблоне быть не может, то есть каждый параметр, определенный в операторе template, обязательно должен использоваться в определении шаблона.

Ловушка: сложности компиляции

Некоторые компиляторы не поддерживают отдельной компиляции шаблонов, поэтому шаблон приходится включать в программный код, в котором он используется. Обычно применению шаблона должно предшествовать его объявление.

Надежнее всего не пользоваться объявлениями шаблонов функций, а включать определение шаблона в тот файл, где он применяется, причем до первого вызова определяемой им функции. Поскольку один и тот же шаблон может использоваться в нескольких файлах, его следует включать в файлы не непосредственно, а с помощью директивы #include. Иными словами, определение шаблона функции следует записать в один-единственный файл, а в те файлы, где он используется, включать директиву #include с именем файла шаблона.

Некоторые компиляторы C++ предъявляют к использованию шаблонов дополнительные требования. Если с компиляцией возникнут проблемы, обратитесь к документации компилятора или спросите совета у специалиста. Возможно, потребуется установить специальные настройки или реорганизовать определения шаблонов и других элементов в файлах.

Шаблон функции

Определение и объявление шаблона функции начинаются следующей строкой:

```
template<class параметр_типа>
```

Объявление (если оно используется) и определение шаблона функции ничем не отличаются от объявления и определения обычной функции за исключением использования параметра типа шаблона вместо обычного имени типа параметров функции.

Например, следующее объявление является объявлением шаблона функции:

```
template<class T>
void show_stuff(int stuff1, T stuff2, T stuff3);
```

Ему может соответствовать приведенное ниже определение шаблона функции:

```
template<class T>
void show_stuff(int stuff1, T stuff2, T stuff3)
{
    cout << stuff1 << endl
         << stuff2 << endl
         << stuff3 << endl;
}
```

Шаблон из этого примера эквивалентен целой группе объявлений и определений функций — по одному для каждого возможного имени типа. Имя типа подставляется вместо имени параметра — в данном примере вместо идентификатора `T`. Рассмотрим такой вызов:

```
show_stuff(2, 3.3, 4.4);
```

При его обработке компилятор использует определение функции, полученное путем замены идентификатора `T` именем типа `double`. Для каждого типа данных, с которым используется этот шаблон, генерируется отдельное определение функции, но оно не генерируется ни для одного из типов, с которыми шаблон не используется.

Алгоритмическая абстракция

При описании функции `swap_values` говорилось о применимом к переменным любого типа универсальном алгоритме, который можно использовать для того, чтобы менять местами значения двух переменных. Язык C++ позволяет реализовать такой алгоритм с помощью шаблона функций. Это простейший пример *алгоритмической абстракции*. Когда мы говорим об использовании алгоритмической абстракции, это означает, что алгоритм выражен в общих терминах, и можно игнорировать детали его конкретного применения и сосредоточиться на важнейших частях. Шаблоны функции являются элементами, с помощью которых в C++ поддерживается алгоритмическая абстракция.

Упражнения для самопроверки

1. Напишите шаблон функции `maximum`. Эта функции принимает два значения одного типа и возвращает большее (или любое из них, если они равны). Приведите и объявление, и определение шаблона. В определении должен использоваться оператор `<`, поэтому шаблон будет применим только к тем типам, для которых определен названный оператор. Напишите комментарий к объявлению функции, поясняющий это ограничение.

2. Для определения абсолютного значения числа мы пользовались тремя разными функциями: abs, labs и fabs, отличающимися только типом аргумента. Гораздо удобнее было бы иметь шаблон функции для определения абсолютного значения любого числа. Напишите шаблон такой функции, назвав ее absolute. Он будет применим только к тем типам данных, для которых определен оператор <, а также унарный оператор -, и значения которых можно сравнивать с константой 0. Таким образом, функция absolute может вызываться для любых числовых типов, таких как int, long и double. Приведите и объявление, и определение шаблона функции.
3. Опишите технологию определения шаблонов функций в C++.
4. Какой переменой является параметр T, если префикс шаблона таков:

   ```
   template <class T>
   ```

   а) T должен быть классом;

   б) T не должен быть классом;

   в) T может быть любым встроенным типом языка C++;

   г) T может быть любым типом, встроенным в язык C++ или определенным программистом.

Пример: универсальная функция сортировки

В главе 10 приводился простой алгоритм сортировки массива значений типа int. Этот алгоритм был реализован на C++ в виде функции sort, приведенной в листинге 10.12. Повторим это определение:

```
void sort(int a[], int number_used)
{
    int index_of_next_smallest;
    for (int index = 0; index < number_used-1; index++)
    {   // Помещаем в a[index] правильное значение:
        index_of_next_smallest =
                    index_of_smallest(a, index, number_used);
        swap_values(a[index], a[index_of_next_smallest]);
        // a[0] <= a[1] <=...<= a[index] - это наименьшие и уже
        // отсортированные элементы исходного массива.
        // Остальные элементы пока на своих позициях.
    }
}
```

Внимательно его проанализировав, вы увидите, что базовый тип массива не имеет в нем сколько-нибудь определяющего значения. Если заменить базовый тип в заголовке функции типом double, получится функция сортировки для массивов значений типа double. Конечно, придется подкорректировать и вспомогательные функции, чтобы они тоже могли работать с элементами массивов данного типа. Поэтому рассмотрим эти функции, вызываемые из функции sort. Их две: swap_values и index_of_smallest.

Мы уже видели, что первая из них применима к переменным любого типа, если только определить ее как шаблон функции (см. листинг 14.1). А зависит ли функция

index_of_smallest от базового типа сортируемого массива? Вот как мы ее определили в главе 10:

```
int index_of_smallest(const int a[], int start_index, int number_used)
{
    int min = a[start_index];
    int index_of_min = start_index;
    for (int index = start_index+1;
              index < number_used; index++)
        if (a[index] < min)
        {
            min = a[index];
            index_of_min = index;
            // min – наименьший из элементов
            // от a[start_index] до a[index].
        }

    return index_of_min;
}
```

Эта функция тоже не зависит от базового типа массива. Если заменить два выделенных вхождения имени типа int типом double, получится новая версия функции index_of_smallest, применимая к массивам базового типа double.

Таким образом, чтобы функцию sort можно было использовать для сортировки массивов базового типа double, достаточно заменить несколько вхождений имени типа int типом double. Более того, во втором типе данных тоже нет ничего особенного. Такая же замена возможна и для многих других типов, главное, чтобы для базового типа данных массива был определен оператор <. Это один из тех случаев, когда удобно применять шаблон, а не обычное определение функции. Если в функциях sort и index_of_smallest заменить несколько вхождений имени типа int параметром типа, функция sort сможет сортировать массивы значений любого типа, допускающего сравнение с помощью оператора <. Шаблон, который получится в результате, приведен в листинге 14.2.

Листинг 14.2. Универсальная функция сортировки

```
// Это файл sortfunc.cpp

template<class T>
void swap_values(T& variable1, T& variable2)
    ... Остальная часть определения функции swap_values приведена в листинге 14.1. ...

template<class BaseType>
int index_of_smallest(const BaseType a[], int start_index, int number_used)
{
    BaseType min = a[start_index];
    int index_of_min = start_index;

    for (int index = start_index+1; index < number_used; index++)
        if (a[index] < min)
        {
            min = a[index];
            index_of_min = index;
```

```
            // min – наименьший из элементов от a[start_index] до a[index]
        }
    return index_of_min;
}

template<class BaseType>
void sort(BaseType a[], int number_used)
{
int index_of_next_smallest;
for(int index = 0; index < number_used-1; index++)
   {   // Помещаем в a[index] правильное значение:
         index_of_next_smallest =
              index_of_smallest(a, index, number_used);
         swap_values(a[index], a[index_of_next_smallest]);
        // a[0] <= a[1] <=...<= a[index] – наименьшие и уже
        // отсортированные элементы исходного массива.
        // Остальные элементы пока на своих позициях.
   }
}
```

Обратите внимание, что шаблон функции sort, приведенный в листинге 14.2, может использоваться с массивами значений, вообще не являющихся числами. В демонстрационной программе листинга 14.3 эта функция вызывается для сортировки массива символов. Символы можно сравнивать с помощью оператора <, и хотя в отношении символов значение этого оператора может немного меняться в зависимости от реализации, в основном он будет действовать так же, как для чисел. Например, если сравнить с его помощью две заглавные буквы, он вернет значение, указывающее, предшествует ли в алфавите первая буква второй. То же самое он вернет и при сравнении двух букв нижнего регистра. Если смешать буквы верхнего и нижнего регистра, ситуация будет не такой однозначной. Программа, приведенная в листинге 14.3, работает только с символами верхнего регистра. Здесь массив букв верхнего регистра сортируется по алфавиту с помощью вызова шаблона функции sort. (Этот шаблон отсортирует даже массив объектов определенного программистом класса, если только этот класс содержит перегруженную версию оператора <.)

Листинг 14.3. Использование универсальной функции сортировки

```
// Демонстрирует универсальную функцию сортировки.
#include <iostream>
using namespace std;

// В файле sortfunc.cpp определена следующая функция:

// template<class BaseType>
// void sort(BaseType a[], int number_used);

// Многие компиляторы позволяют включить это объявление шаблона функции как объявление,
// а не как комментарий. Однако можно обойтись и без него, поскольку определение функции
// находится в файле sortfunc.cpp, включенном в данный файл с помощью директивы include,
// и потому оно находится перед функцией main.
```

продолжение ⇩

Листинг 14.3 (*продолжение*)

```
// Предусловие: number_used определяет размер массива a.
// Элементы массива от a[0] до a[number_used-1] содержат значения.
// Постусловие: значения элементов от a[0] до a[number_used-1] упорядочены
// таким образом, что a[0] <= a[1] <= ... <= a[number_used-1].

#include "sortfunc.cpp"

int main()
{
    int i;
    int a[10] = {9, 8, 7, 6, 5, 1, 2, 3, 0, 4};
    cout << "Unsorted integers:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;
    sort(a, 10);
    cout << "In sorted order the integers are:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;

    double b[5] = {5.5, 4.4, 1.1, 3.3, 2.2};
    cout << "Unsorted doubles:\n";
    for (i = 0; i < 5; i++)
        cout << b[i] << " ";
    cout << endl;
    sort(b, 5);
    cout << "In sorted order the doubles are:\n";
    for (i = 0; i < 5; i++)
        cout << b[i] << " ";
    cout << endl;

    char c[7] = {'G', 'E', 'N', 'E', 'R', 'I', 'C'};
    cout << "Unsorted characters:\n";
    for (i = 0; i < 7; i++)
        cout << c[i] << " ";
    cout << endl;
    sort(c, 7);
    cout << "In sorted order the characters are:\n";
    for (i = 0; i < 7; i++)
        cout << c[i] << " ";
    cout << endl;

    return 0;
}
```

Вывод

```
Unsorted integers:
9 8 7 6 5 1 2 3 0 4
In sorted order the integers are:
0 1 2 3 4 5 6 7 8 9
Unsorted doubles:
5.5 4.4 1.1 3.3 2.2
In sorted order the doubles are:
1.1 2.2 3.3 4.4 5.5
```

```
Unsorted characters:
G E N E R I C
In sorted order the characters are:
C E E G I N R
```

Совет программисту: как определять шаблоны

Определяя шаблон функции, приведенный в листинге 14.2, мы начали с функции, сортирующей элементы массива типа int. Затем на ее основе создали шаблон, заменив базовый тип данных массива параметром типа T. Это удобный подход к написанию шаблонов. Когда вам потребуется шаблон функции, напишите сначала версию этой функции для какого-нибудь конкретного типа данных, полностью отладьте ее, а уже затем преобразуйте в шаблон, заменив некоторые имена типов параметрами типов. У описанного подхода имеются два преимущества. Во-первых, определяя обычную функцию, вы имеете дело с более конкретным случаем, что облегчает визуализацию задачи. Во-вторых, на каждой стадии разработки понадобится контролировать меньшее количество деталей; в частности, занимаясь алгоритмом, не нужно будет думать о правилах синтаксиса шаблонов.

Ловушка: использование шаблона с неподходящим типом данных

Шаблон функции можно применять с любым типом данных, для которого имеет смысл код, который в нем содержится. При этом важно, чтобы имел смысл весь код шаблона и чтобы он действовал соответственно значению этого типа данных. Например, шаблон функции swap_values (см. листинг 14.1) нельзя применять с параметрами, для которых оператор присваивания либо не работает вообще, либо работает «неправильно». В частности, недопустимо использование этого шаблона следующим образом:

```
int a[10], b[10];
... Некоторый код, заполняющий массив. ...
swap_values(a, b);
```

Приведенный код не будет работать, поскольку оператор присваивания не работает с аргументами типа массивов.

Упражнения для самопроверки

5. В листинге 10.10 главы 10 приведено определение функции search, выполняющей поиск в массиве заданного значения типа int. Создайте шаблон этой функции, применимый к массивам элементов любого типа. Напишите и объявление, и определение данного шаблона. Подсказка. Он будет почти идентичен функции в листинге 10.10.
6. В практическом задании 9 главы 3 вам предлагалось перегрузить функцию abs для работы с несколькими встроенными типами, изученными к тому моменту. Сравните перегрузку данной функции с использованием для этой же цели шаблона, определенного в упражнении 2.

14.2. Шаблоны и абстракция данных

Равные доходы и равные культурные возможности...
сделали нас всех членами одного класса.
Эдвард Беллами

В предыдущем разделе рассказывалось, что определение функции можно сделать более универсальным с помощью шаблона. Теперь же мы поговорим о том, как с помощью шаблонов создавать универсальные определения классов.

Синтаксис шаблона класса

У шаблонов классов практически тот же синтаксис, что и у шаблонов функций. Перед определением шаблона помещается строка

```
template<class T>
```

Параметр типа T используется в определении класса так же, как любой другой тип данных. Он представляет произвольный тип, причем не обязательно тип класса. Как и в шаблоне функции, вместо идентификатора T для параметра типа может применяться любой другой идентификатор.

Рассмотрим следующий шаблон класса:

```
// Класс для пары значений типа T.
template<class T>
class Pair
{
public:
    Pair();

    Pair(T first_value, T second_value);

    void set_element(int position, T value);
    // Предусловие: значение аргумента position равно 1 или 2.
    // Постусловие: переменной, номер которой определяет
    // аргумент position, присвоено значение value.

    T get_element(int position) const;
    // Предусловие: значение аргумента position равно 1 или 2.
    // Возвращает значение переменной, номер которой
    // определяет аргумент position.
private:
    T first;   // Первая переменная.
    T second;  // Вторая переменная.
};
```

Объект этого класса содержит пару значений типа T; если T — это int, объект содержит пару целых чисел, если T — это char, то пару символов, и т. д.

После определения шаблона класса можно объявлять объекты этого класса. В объявлении должно быть указано, какой тип следует подставить вместо T. Например,

следующие два оператора объявляют объект score для хранения пары целых чисел и объект seats для хранения пары символов:

```
Pair<int> score;
Pair<char> seats;
```

Эти объекты могут использоваться как любые другие объекты программы. Так, приведенный далее код присваивает объекту score, представляющему счет игры двух команд, 3 очка для первой команды и 0 очков для второй:

```
score.set_element(1, 3);
score.set_element(2, 0);
```

Функции-члены шаблона класса определяются так же, как и функции-члены обычных классов. Единственным их отличием является то, что их определения сами являются шаблонами. Вот определения функции-члена set_element и конструктора с двумя аргументами:

```
// Используем библиотеки классов iostream и cstdlib:
template<class T>
void Pair<T>::set_element(int position, T value)
{
    if (position == 1)
        first = value;
    else if (position == 2)
        second = value;
    else
    {
        cout << "Error: Illegal pair position.\n";
        exit(1);
    }
}

template<class T>
Pair<T>::Pair(T first_value, T second_value)
           : first(first_value), second(second_value)
{
    // Тело функции намеренно оставлено пустым.
}
```

Обратите внимание, что именем класса перед оператором доступа к члену класса является Pair<T>, а не просто Pair.

Имя шаблона класса может использоваться в качестве типа параметра функции. Например, допустимо следующее объявление функции, параметром которой является объект, представляющий пару целых чисел:

```
int add_up(const Pair<int>& the_pair);
// Возвращает сумму двух целых чисел, хранящихся в объекте the_pair.
```

Обратите внимание, что в объявлении параметра задан тип данных, который должен быть подставлен вместо параметра T.

Шаблон класса можно использовать даже в шаблоне функции. Так, вместо определения объявленной выше специализированной функции add_up можно определить шаблон функции, применимый к числам любого типа:

```
template<class T>
T add_up(const Pair<T>& the_pair);
```

```
// Предусловие: для значений типа T определен оператор +.
// Возвращает сумму двух значений, хранящихся в объекте the_pair.
```

Синтаксис шаблона класса

Определение класса и определения его функций-членов предваряются следующей строкой:

`template<class` *параметр_типа*`>`

В остальном названные определения такие же, как у обычного класса, с той разницей, что вместо конкретного типа данных используется параметр типа.

Далее приведено начало определения шаблона класса:

```
template<class T>
class Pair
{
public:
    Pair();
    Pair(T first_value, T second_value);
    void set_element(int position, T value);
        ...
```

Функции-члены и перегруженные операторы определяются как шаблоны функций. Например, определение функции-члена для приведенного шаблона класса может начинаться так:

```
template<class T>
void Pair<T>::set_element(int position, T value)
{
    ...
```

Определения типов

Шаблон класса можно специализировать, задав для него в аргументе тип данных, как в следующем примере:

`Pair<int>`

Специализированное имя класса, подобное `Pair<int>`, можно использовать наравне с именами любых других классов. В частности, допустимо его применение для объявления объектов или задания типа формальных параметров функций.

Можно определить имя класса с тем же значением, что у специализированного имени шаблона класса, такого как `Pair<int>`. Определение нового имени типа класса имеет следующий синтаксис:

`typedef` *имя_класса<аргумент_типа> новое_имя_типа*`;`

Например:

```
typedef Pair<int> PairOfInt;
```

После этого имя типа `PairOfInt` может использоваться для объявления объектов типа `Pair<int>`, как в приведенном ниже примере:

```
PairOfInt pair1, pair2;
```

Кроме того, оно может применяться для определения типа формальных параметров.

Пример: класс-массив

В листинге 14.4 приведен интерфейс шаблона класса, объекты которого представляют списки значений. Поскольку это определение является шаблоном класса, списки могут состоять из элементов любого типа. Скажем, можно создавать объекты, содержащие списки значений типа int, типа double, типа string и т. д.

Пример демонстрационной программы, в которой используется этот шаблон, содержится в листинге 14.5. Освоив детали синтаксиса, вы сможете применять данный шаблон класса в любой программе, где требуется список значений. Реализация шаблона класса показана в листинге 14.6.

Обратите внимание, что мы перегрузили оператор вывода так, чтобы с его помощью можно было выводить данные объектов шаблона класса List. Для этого оператор << сделан дружественной функцией класса. Чтобы его параметр имел такой же тип, как класс, мы определили его тип как List<ItemType>. Например, когда параметр типа заменяется типом int, тип параметра оператора << заменяется типом List<int>.

Листинг 14.4. Интерфейс шаблона класса List

```
// Это заголовочный файл list.h. В нем содержится интерфейс класса List.
// Объекты типа List могут содержать списки элементов любого типа,
// для которого определены операторы  << и =. Однако все элементы
// каждого списка должны принадлежать к одному типу.
// Список элементов типа Type_Name максимальной длины объявляется так:
//     List<Type_Name> the_object(max);
#ifndef LIST_H
#define LIST_H
#include <iostream>
using namespace std;

namespace listsavitch
{
    template<class ItemType>
    class List
{
    public:
        List(int max);
        // Инициализирует объект пустым списком, вмещающим
        // до max элементов типа ItemType.

        ~List();
        // Возвращает всю используемую объектом динамическую память
        // в динамическую область.

        int length() const;
        // Возвращает количество элементов в списке.

        void add(ItemType new_item);
        // Предусловие: список не полон.
        // Постусловие: в список добавлено значение аргумента new_item.

        bool full() const;
        // Возвращает true, если список полон.
```

продолжение ⇩

Листинг 14.4 (*продолжение*)

```
        void erase();
        // Очищает список, удаляя из него все элементы.

        friend ostream& operator <<(ostream& outs,
                                    const List<ItemType>& the_list);
        // Перегружает оператор  <<, чтобы им можно было
        // пользоваться для вывода содержимого списка.
        // Каждый элемент выводится с новой строки.
        // Предусловие: если outs – выходной файловый поток,
        // он уже соединен с файлом.
    private:
        ItemType *item;   // Указатель на динамический массив,
                          // в котором хранится список.
        int max_length;   // Максимальное текущее количество
                          // элементов списка.
        int current_length;   // Текущее количество элементов списка.
    };
}   // listsavitch
#endif //LIST_H
```

Листинг 14.5. Программа, использующая шаблон класса List

```
// Программа, демонстрирующая использование шаблона класса List.
#include <iostream>
#include "list.h"
#include "list.cpp"  // Благодаря включению файла list.cpp достаточно откомпилировать
                     // только данный файл (содержащий функцию main).
using namespace std;
using namespace listsavitch;

int main()
{
List<int> first_list(2);
first_list.add(1);
    first_list.add(2);
    cout << "first_list = \n"
         << first_list;

List<char> second_list(10);
second_list.add('A');
    second_list.add('B');
    second_list.add('C');
    cout << "second_list = \n"
         << second_list;

    return 0;
}
```

Вывод

```
first_list =
1
2
second_list =
A
B
C
```

Листинг 14.6. Реализация класса List

```
// Это файл реализации: list.cpp.
// В нем содержится реализация шаблона класса List.
// Интерфейс данного шаблона класса находится в заголовочном файле list.h.
#ifndef LIST_CPP
#define LIST_CPP
#include <iostream>
#include <cstdlib>
#include "list.h"   // Хотя в данном файле это не требуется,
                    // использование директивы #ifndef в файле list.h полезно
                    // для тех случаев, когда шаблон класса
                    // применяется в нескольких файлах программы.
using namespace std;

namespace listsavitch
{
    // Используем библиотеку классов cstdlib:
    template<class ItemType>
    List<ItemType>::List(int max) : max_length(max), current_length(0)

    {
          item = new ItemType[max];
    }

    template<class ItemType>
    List<ItemType>::~List()
    {
        delete [] item;
    }

    template<class ItemType>
    int List<ItemType>::length() const
    {
        return (current_length);
    }

    // Используем библиотеки классов iostream и cstdlib:
    template<class ItemType>
    void List<ItemType>::add(ItemType new_item)
    {
        if (full())
        {
            cout << "Error: adding to a full list.\n";
            exit(1);
        }
        else
        {
            item[current_length] = new_item;
            current_length = current_length + 1;
        }
    }

    template<class ItemType>
    bool List<ItemType>::full() const
```

продолжение

Листинг 14.6 (*продолжение*)

```
    {
        return (current_length == max_length);
    }

    template<class ItemType>
    void List<ItemType>::erase()
    {
        current_length = 0;
    }

        // Используем библиотеку классов iostream:
    template<class ItemType>
    ostream& operator <<(ostream& outs, const List<ItemType>& the_list)
    {
        for (int i = 0; i < the_list.current_length; i++)
            outs << the_list.item[i] << endl;

        return outs;
    }
}   // listsavitch
#endif   // LIST_CPP (Обратите внимание, что все определения шаблона
         // располагаются между директивами #ifndef... #endif.)
```

Упражнения для самопроверки

7. Приведите определение функции-члена get_element для шаблона класса Pair, описанного в разделе «Синтаксис шаблона класса».
8. Приведите определение конструктора без аргументов для шаблона класса Pair, описанного в разделе «Синтаксис шаблона класса».
9. Приведите определение шаблона класса HeterogeneousPair, похожего на шаблон класса Pair, описанного в разделе «Синтаксис шаблона класса», но отличающегося тем, что его первая и вторая переменные могут содержать значения разных типов. Используйте два типа параметров с именами T1 и T2. Первая переменная-член класса должна иметь тип T1, а вторая — T2. Единственная функция-мутатор set_element шаблона класса Pair в шаблоне класса HeterogeneousPair должна быть заменена двумя мутаторами: set_first и set_second. Точно так же единственная функция-аксессор get_element шаблона класса Pair в шаблоне класса HeterogeneousPair должна быть заменена двумя аксессорами: get_first и get_second.
10. Верно ли следующее утверждение: дружественные функции одинаково используются и для шаблонов классов, и для классов?

Резюме

- С помощью шаблонов функций можно определять функции с параметрами для типов их аргументов.
- Классы с параметрами для типов их членов также можно определять с помощью шаблонов функций.

Ответы к упражнениям для самопроверки

1. Объявление функции:

```
template<class T>
T maximum(T first, T second);
// Предусловие: для типа T определен оператор <.
// Возвращает максимальные значения аргументов first и second.
```

Определение:

```
template<class T>
T maximum(T first, T second)
{
    if (first < second)
        return second;
    else
        return first;
}
```

2. Объявление функции:

```
template<class T>
T absolute(T value);
// Предусловие: выражения x < 0 и -x определены
// для x типа T.
// Возвращает абсолютное значение аргумента.
```

Определение:

```
template<class T>
T absolute(T value)
{
    if (value < 0)
        return -value;
    else
        return value;
}
```

3. Шаблоны обеспечивают возможность определять функции и классы с параметрами для имен типов.
4. Ответ г). T может быть любым типом, как примитивным (встроенным в язык C++), так и определенным программистом (тип class, struct или enum, тип определенный для массива и т. д.).
5. Объявление и определение функции приведены ниже. Они практически идентичны версиям листинга 10.10 из главы 10, с той разницей, что два вхождения имени типа int заменены в списке параметров идентификатором BaseType.

Объявление функции:

```
template<class BaseType>
int search(const BaseType a[],
                    int number_used, BaseType target);
// Предусловие: number_used <= объявленного размера массива a.
// Элементы от a[0] до a[number_used-1] содержат значения.
```

```
// Возвращает первый индекс, для которого a[index] == target,
// если таковой имеется; в противном случае возвращает -1.
```

Определение:

```
template<class BaseType>
int search(const BaseType a[], int number_used,
                                        BaseType target)
{
    int index = 0, found = false;
    while ((!found) && (index < number_used))
        if (target == a[index])
            found = true;
        else
            index++;

    if (found)
        return index;
    else
        return -1;
}
```

6. Перегрузка функций работает только для тех типов, для которых она определена. Она может работать и тогда, когда выполняется автоматическое приведение одного типа к другому, но при этом результат может оказаться не таким, как ожидалось. Шаблон работает для любого заданного на момент вызова типа, если только для него определен оператор <.

7.
```
// Используем библиотеки классов iostream и cstdlib:
template<class T>
T Pair<T>::get_element(int position) const
{
    if (position == 1)
        return first;
    else if (position == 2)
        return second;
    else
    {
        cout << "Error: Illegal pair position.\n";
        exit(1);
    }
}
```

8. Поскольку подходящих инициализационных значений по умолчанию не имеется, этот конструктор ничего не делает, но он позволяет объявлять (инициализировать) объекты, не задавая аргументов для конструктора.

```
template<class T>
Pair<T>::Pair()
{
    // Ничего не делать.
}
```

9.
```
// Класс для пары значений, first типа T1
// и second типа T2:
template<class T1, class T2>
class HeterogeneousPair
```

```
{
public:
    HeterogeneousPair();

    HeterogeneousPair(T1 first_value, T2 second_value);

    void set_first(T1 value);

    void set_second(T2 value);

    T1 get_first() const;

    T2 get_second() const;

private:
    T1 first;
    T2 second;
};
```

Определения функций-членов таковы:

```
template<class T1, class T2>
HeterogeneousPair<T1, T2>::HeterogeneousPair()
{
    // Ничего не делать.
}

template<class T1, class T2>
HeterogeneousPair<T1, T2>::HeterogeneousPair
     (T1 first_value, T2 second_value)
           : first(first_value), second(second_value)
{
    // Тело функции намеренно оставлено пустым.
}


template<class T1, class T2>
T1 HeterogeneousPair<T1, T2>::get_first() const
{
    return first;
}


template<class T1, class T2>
T2 HeterogeneousPair<T1, T2>::get_second() const
{
    return second;
}


template<class T1, class T2>
void HeterogeneousPair<T1, T2>::set_first(T1 value)
{
    first = value;
}


template<class T1, class T2>
```

```
void HeterogeneousPair<T1, T2>::set_second(T2 value)
{
    second = value;
}
```

10. Да.

Практические задания

1. Напишите шаблон для функции, одним параметром которой является частично заполненный массив, а вторым — значение базового типа массива. Если значение второго параметра присутствует в массиве, функция возвращает индекс первой индексированной переменной, которая его содержит; в противном случае она возвращает -1. Базовый тип массива является параметром типа. Обратите внимание, что для задания частично заполненного массива требуется два параметра: один для самого массива, а второй для количества его заполненных элементов. Напишите отладочную программу, чтобы протестировать этот шаблон.
2. Перепишите определение шаблона класса `List`, приведенное в листингах 14.4 и 14.6, чтобы сделать его еще более универсальным. Эта новая версия должна позволять по очереди перебирать элементы списка. В каждый момент времени один из элементов должен быть текущим. Всегда можно запросить текущий элемент, сделать текущим следующий или предыдущий элемент, начать с начала списка, сделав текущим первый элемент, а также запросить n-й элемент списка. Для этого в шаблон класса нужно добавить следующие члены: переменную для хранения позиции текущего элемента в списке; функцию, возвращающую текущий элемент; функцию, делающую текущим следующий элемент; функцию, делающую текущим предыдущий элемент; функцию, делающую текущим первый элемент; функцию, возвращающую n-й элемент списка (получающую значение n в качестве аргумента). (Элементы нумеруются так же, как в массивах, то есть первый элемент имеет номер 0, второй — 1, третий — 2 и т. д.)

   Обратите внимание, что иногда некоторые из указанных действий невозможны. Например, пустой список не имеет первого элемента, и ни в одном списке нет элемента, следующего за последним. Поэтому подобные условия должны обязательно проверяться и правильно обрабатываться. Напишите программу для отладки этого шаблона класса.
3. Напишите шаблон функции с параметрами для списка элементов и, возможно, присутствующего в этом списке элемента. Если элемент имеется в списке, функция возвращает позицию его первого вхождения, в противном случае — значение -1. Первая позиция в списке — 0, следующая — 1 и т. д. Параметром типа является тип элементов списка. Воспользуйтесь шаблоном класса `List`, определенным в практическом задании 2. Напишите программу для отладки шаблона функции.

4. Выполните практическое задание 2 из главы 10 и сделайте функцию `delete_repeats` шаблоном функции с параметром типа для базового типа массива. Будет проще сначала выполнить задание из главы 10, а затем его модифицировать.
5. В листинге 14.3 приведен шаблон функции для сортировки массива с использованием алгоритма сортировки выборкой. Напишите подобный шаблон для сортировки массива с использованием алгоритма сортировки вставкой, описанного в практическом задании 5 главы 10. Если вы еще не выполняли это задание, будет проще сначала сделать его, а затем превратить функцию в шаблон.
6. Напишите шаблон функции для итеративного двоичного поиска на основе функции, приведенной в листинге 13.6. Сформулируйте требования к параметру типа шаблона.
7. Напишите шаблон функции для рекурсивного двоичного поиска на основе функции, приведенной в листинге 13.7. Сформулируйте требования к параметру типа шаблона.

Глава 15 Указатели и связные списки

Кто любовью чистой полон,
На того укажет сердце,
И тебе, мой друг любезный,
Укажу я на него.

Гилберт и Салливан

Связный список — это список, составленный с использованием указателей. Его размер не фиксирован, он может увеличиваться и уменьшаться во время выполнения программы. В этой главе рассказывается, каким образом определять связные списки и как ими манипулировать. Это новый для вас и весьма полезный способ применения указателей.

15.1. Узлы и связные списки

Необходимые программе динамические переменные редко бывают таких простых типов, как `int` и `double`, — обычно они относятся к сложным типам вроде массивов, структур или классов. Вы уже убедились, насколько полезны динамические переменные типа массивов. Можно также использовать динамические переменные типа структур или классов, но они обычно имеют одну или несколько переменных-членов типа указателей, связывающих их с другими динамическими переменными. В качестве примера на рис. 15.1 показана структура, представляющая собой список покупок.

Узлы

Приведенная на рис. 15.1 конструкция состоит из последовательности элементов, изображенных в виде соединенных стрелками прямоугольников. Эти прямоугольники называются *узлами*, а стрелки представляют указатели. Каждый узел на рисунке содержит строку, целое число и указатель, который может ссылаться на

другой узел того же типа. Обратите внимание, что указатели ссылаются на весь узел, а не на его отдельные элементы (такие, как 10 или "rolls").

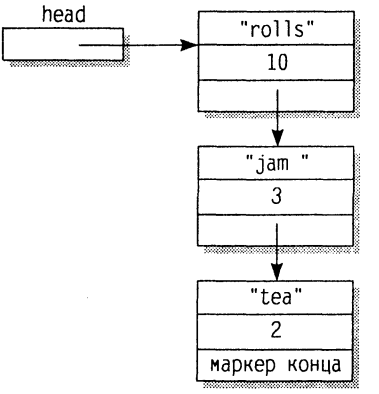


Рис. 15.1. Узлы и указатели

Узлы реализуются как структуры или классы C++. Например, ниже приведено определение структуры для узла конструкции, показанной на рис. 15.1, и определение типа указателя на такие узлы:

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};

typedef ListNode* ListNodePtr;
```

В данном случае важен порядок определений типов. Первым должен быть определен тип ListNode, поскольку он используется в определении типа ListNodePtr.

Прямоугольник, помеченный на рисунке словом «head», — это не узел, а просто переменная-указатель. Она объявляется следующим образом:

```
ListNodePtr head;
```

И хотя мы расположили определения типов так, чтобы избежать цикличности, приведенное выше определение структуры ListNode само по себе циклично, так как для определения переменной-члена link в нем используется тип ListNode. Но в такой цикличности нет ничего противоречащего правилам, и C++ ее допускает.

Теперь у нас имеются указатели в составе структур, которые ссылаются на структуры, содержащие указатели. Синтаксис этих конструкций может быть немного громоздким, но он четко соответствует правилам обращения с указателями и структурами. Предположим, что нам нужно заменить значение 10 в первом узле конструкции, показанной на рис. 15.1, значением 12. Это можно сделать с помощью следующего оператора:

```
(*head).count = 12;
```

Выражение слева от оператора присваивания нуждается в пояснении. Поскольку переменная head является указателем, выражение *head представляет то, на что он указывает, — узел (динамическую переменную), который содержит значения "rolls" и 10. Узел, на который ссылается *head, — это структура, а переменная-член этой структуры, содержащая значение типа int, называется count. Поэтому (*head).count является ссылкой на переменную типа int первого узла. Скобки, в которые заключено выражение *head, обязательны. Дело в том, что оператор разыменования (*) в данном случае должен быть выполнен до оператора доступа к члену структуры (.), однако последний имеет более высокий приоритет, и поэтому без скобок он выполнится первым, что приведет к ошибке. Ниже описан сокращенный синтаксис, позволяющий обойтись без скобок.

В языке C++ предусмотрен оператор, предназначенный для использования с указателями и очень упрощающий обращение к членам структуры или класса. Это *оператор косвенного доступа к члену класса или структуры*, иначе называемый *оператором стрелка* (->).

Оператор стрелка (оператор косвенного доступа к члену структуры или класса)

Оператор стрелка (->) служит для доступа к члену структуры (или объекта класса), на которую (который) указывает переменная-указатель. Его синтаксис таков:

переменная_указатель->имя_члена

Это выражение является ссылкой на член *имя_члена* структуры или объекта, на которую (который) указывает *переменная_указатель*. Предположим, что у нас имеется следующее определение:

```
struct Record
{
    int number;
    char grade;
};
```

Тогда приведенный ниже код создает динамическую переменную типа Record и присваивает ее переменным-членам значения 2001 и 'A':

```
Record *p;
p = new Record;
p->number = 2001;
p->grade = 'A';
```

Сочетая действия операторов звездочка и точка, он служит для указания члена динамической структуры или объекта, на которую (который) ссылается указатель. Например, с его помощью можно переписать приведенный выше оператор присваивания, изменяющий числовое значение в первом узле, так:

```
head->count = 12;
```

Эти два оператора эквивалентны, но в программах, конечно же, используется вторая форма.

Строка в первом узле нашей конструкции может быть заменена строкой "bagels" с помощью следующего оператора:

```
head->item = "bagels";
```

Результат этих изменений в первом узле списка показан на рис. 15.2.

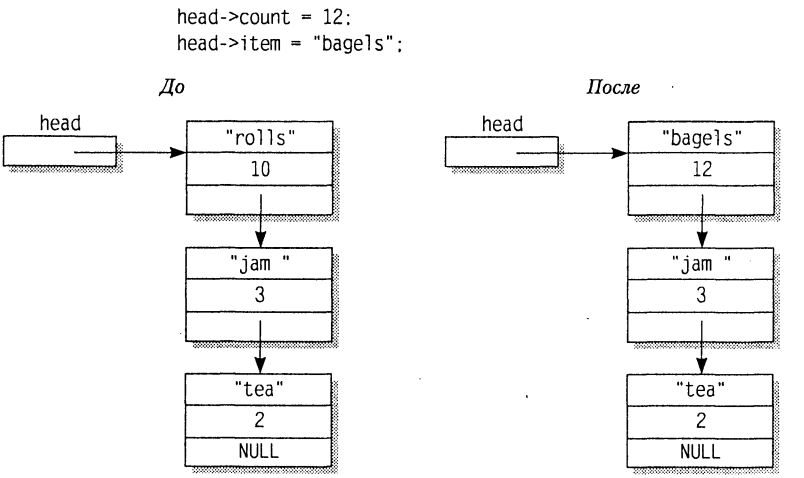


Рис. 15.2. Доступ к данным узла

Посмотрите на указатель в последнем узле списков, показанных на рисунке, — там, где должен располагаться указатель, записано слово NULL. На рис. 15.1 в этом месте было написано «маркер конца», что не является выражением C++. В программах на этом языке в качестве маркера конца используется специальная константа NULL, являющаяся частью языка и определенная в одной из обязательных стандартных библиотек.

Константа NULL выполняет две разные функции. Во-первых, в ее помощью можно присвоить значение переменной-указателю, которая в противном случае не имела бы значения. Тем самым предотвращается случайное обращение к памяти, поскольку NULL не является адресом какой-либо ячейки памяти. Во-вторых, эта константа используется в качестве маркера конца. Когда, обрабатывая узлы списка, показанного на рис. 15.2, программа встретит узел, содержащий NULL, она будет «знать», что достигла конца списка.

На самом деле указанная константа является числом 0, однако мы предпочитаем называть ее NULL и считать не числом, а особым значением, что лучше отвечает ее назначению. Определение NULL имеется в нескольких стандартных библиотеках (например, в iostream и cstddef), поэтому для использования этого идентификатора нужно включить в программу с помощью директивы #include одну из таких библиотек. Директива using в данном случае не нужна, в частности не требуется директива using namespace std;, хотя для других элементов кода программы она может быть необходима.

Указатель устанавливается в значение NULL с помощью оператора присваивания, как в следующем примере:

```
double *there = NULL;
```

Здесь определяется переменная-указатель there, которая инициализируется значением NULL. Константа NULL может быть присвоена переменной-указателю на любой тип данных.

NULL

NULL — это специальное константное значение, используемое для переменных-указателей, которые без него не содержали бы значений. Данную константу можно присвоить переменной-указателю любого типа. Она определяется в нескольких библиотеках, включая библиотеки с заголовочными файлами <iostream> и <cstddef>. На самом деле эта константа является значением 0, но мы предпочитаем называть ее NULL.

Упражнения для самопроверки

1. Предположим, что у вас в программе имеются следующие определения типов:

```
struct Box
{
    string name;
    int number;
    Box *next;
};

typedef Box* BoxPtr;
```

Что будет выводить следующий код:

```
BoxPtr head;
head = new Box;
head->name = "Sally";
head->number = 18;
cout << (*head).name << endl;
cout << head->name << endl;
cout << (*head).number << endl;
cout << head->number << endl;
```

2. Допустим, в вашу программу включены определения типов и код, указанные в упражнении 1. Этот код создает узел, содержащий значение "Sally" типа string и число 18. Какой код нужно добавить, чтобы установить значение переменной-члена next данного узла равным NULL?
3. Предположим, в вашей программе имеются определения типов и код, указанные в упражнении 1. Если значение переменной-указателя head еще не изменено, как удалить динамическую переменную, на которую указывает head, и вернуть занимаемую ею память в динамическую область для повторного использования?

4. Используя следующие определения типов:

```
struct ListNode
{
    string item;
    int count;
    ListNode *link;
};

ListNode *head = new ListNode;
```

напишите код, присваивающий строку "Wilbur's brother Orville" переменной-члену item узла, на который ссылается указатель head.

Связные списки

Списки, подобные приведенному на рис. 15.2, называются связными. *Связный список* — это список узлов, в котором каждый узел содержит переменную-член типа указателя, ссылающуюся на следующий узел списка. Первый узел называется началом списка или *«головой»*, поэтому переменную-указатель на первый элемент списка мы назвали head. Обратите внимание, что переменная-указатель только указывает на первый элемент списка, а не является таковым. Последний узел не имеет специального названия, но зато обладает особым свойством: его переменная-указатель всегда имеет значение NULL. Поэтому, чтобы узнать, является ли некоторый узел последним, достаточно проверить, содержит ли его переменная-указатель это значение.

В настоящем разделе мы напишем несколько базовых функций для работы со связными списками. Чтобы упростить задачу, будем пользоваться простейшими узлами — такими, как на рис. 15.2, содержащими только целое число и указатель. Определения типов структуры и указателя для наших узлов таковы:

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;
```

В качестве подготовительного упражнения сконструируем начало связного списка с узлом данного типа. Сначала объявим переменную-указатель head, которая будет ссылаться на первый элемент списка.

```
NodePtr head;
```

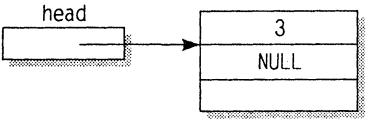
Для создания первого узла потребуется оператор new, создающий новую динамическую переменную — первый узел связного списка:

```
head = new Node;
```

Затем присвоим значения переменным-членам этого узла:

```
head->data = 3;
head->link = NULL;
```

Обратите внимание, что переменная-указатель, входящая в состав узла, устанавливается равной NULL, поскольку пока данный узел является первым и последним в списке. Вот как выглядит наш связной список на этом этапе:



Итак, у нас получился одноэлементный список, созданный произвольным образом. Разумеется, для составления списков большего размера нужно как-то систематизировать процесс добавления узлов. В следующем разделе описывается один из простейших способов добавления узлов в связные списки.

Вставка узла в начало списка

Предположим, что связной список уже содержит один или несколько узлов, и мы должны разработать функцию для добавления еще одного узла. Первым параметром этой функции будет передаваемая по ссылке переменная-указатель, ссылающаяся на первый элемент списка. Вторым параметром будет число, которое нужно записать в новый узел. Объявление нашей функции вставки следующее:

```
void head_insert(NodePtr& head, int the_number);
```

> **Связные списки в качестве аргументов**
>
> Для работы со связным списком требуется наличие переменной, указывающей на его первый элемент. Эта переменная-указатель выполняет роль имени списка. При написании функции, принимающей связной список в качестве аргумента, таким аргументом может служить переменная-указатель на начало списка.

Прежде чем вставлять в связной список новый узел, наша функция сначала должна будет создать этот узел с помощью оператора new. Затем она скопирует в него данные, а уже потом вставит его в список. Вставка будет выполняться таким образом, чтобы новый элемент стал первым элементом списка. Поскольку у динамических переменных нет имен, для доступа к новому узлу будет использоваться локальная переменная-указатель. Если назвать ее temp_ptr, на новый узел можно будет ссылаться так: *temp_ptr. Далее коротко описан весь этот процесс.

Псевдокод для функции head_insert следующий.

1. Создать новую динамическую переменную, на которую будет указывать переменная temp_ptr. (Эта новая динамическая переменная будет новым узлом списка. На этот список можно ссылаться так: *temp_ptr.)
2. Поместить данные в новый узел.
3. Присвоить указателю link нового узла адрес первого узла исходного списка.
4. Присвоить переменной-указателю head адрес нового узла.

На рис. 15.3 показана диаграмма этого алгоритма. Шаги 2 и 3 здесь определены как следующие операторы присваивания:

```
temp_ptr->link = head;
head = temp_ptr;
```

Полное определение функции приведено в листинге 15.1.

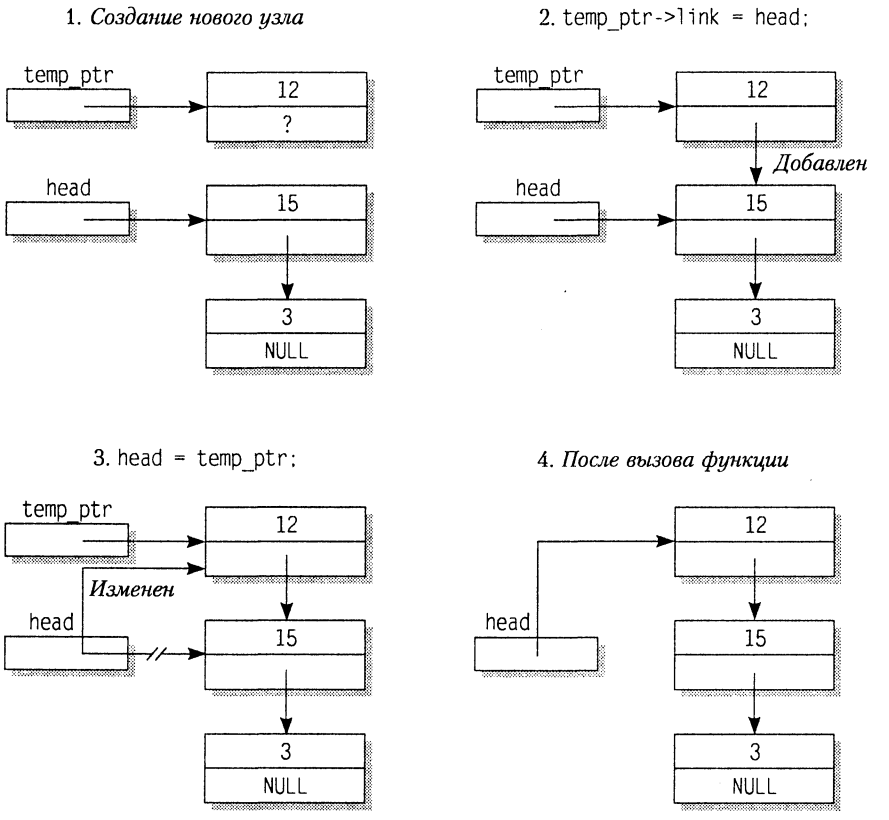


Рис. 15.3. Добавление нового узла в связный список

Листинг 15.1. Функция для добавления узла в начало связного списка

Объявление функции

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;
void head_insert(NodePtr& head, int the_number);
// Предусловие: переменная-указатель head ссылается
// на первый элемент связного списка.
// Постусловие: в начало связного списка добавлен
// новый элемент, содержащий значение the_number.
```

продолжение ⇩

Листинг 15.1 (*продолжение*)

Определение функции

```
void head_insert(NodePtr& head, int the_number)
{
    NodePtr temp_ptr;
    temp_ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = head;
    head = temp_ptr;
}
```

Вероятно, вы захотите, чтобы функция добавления в список нового элемента работала и в том случае, когда в списке не окажется ни одного элемента. Так, список покупок может не содержать ни одного элемента, потому что на этой неделе ничего покупать не нужно. Такой список называется *пустым*. Связный список именуется посредством переменой, указывающей на его первый элемент, но в пустом списке первый элемент отсутствует, и переменная должна содержать значение NULL. Если переменная-указатель head должна ссылаться на первый элемент списка и нужно указать, что этот список пуст, установите значение переменной так:

```
head = NULL;
```

Разрабатывая функцию для операций со связным списком, всегда следует проверять, правильно ли она работает с пустым списком. Если нет, можно добавить специальное условие для этого случая. Если же функция в принципе не может работать с пустыми списками, нужно написать программу так, чтобы она обрабатывала пустые списки каким-либо иным способом, или вообще их не использовать. Однако на практике пустой список обычно может обрабатываться так же, как список, содержащий элементы. Например, функция head_insert, приведенная в листинге 15.1, разрабатывалась для работы с непустым списком, но легко убедиться, что она правильно выполняет свою задачу и для пустого списка.

Ловушка: потерянные узлы

Возможно, вы захотите написать определение функции head_insert таким образом, чтобы она создавала новый узел с использованием переменной-указателя head, а не локальной переменной temp_ptr. Такое определение может начинаться следующим образом:

```
head = new Node;
head->data = the_number;
```

Новый узел создан, содержит корректные данные, и на него, как и положено, указывает переменная head. Теперь нужно только присоединить остальную часть списка, установив указатель нового элемента так, чтобы он указывал на бывший первый элемент списка, то есть

```
head->link
```

Эта ситуация показана на рис. 15.4 (с новым значением 12). Проблема очевидна: после выполнения описанных выше операций на узел, содержащий значение 15,

не указывает ни один указатель, поэтому программа никак не может на него сослаться. Следующий за ним узел тоже потерян. Программа не в состоянии присвоить ни одной из имеющихся у нее переменных-указателей адреса этих двух узлов и не может выполнять с ними никакие другие операции, поскольку у нее просто нет возможности сослаться на эти узлы.

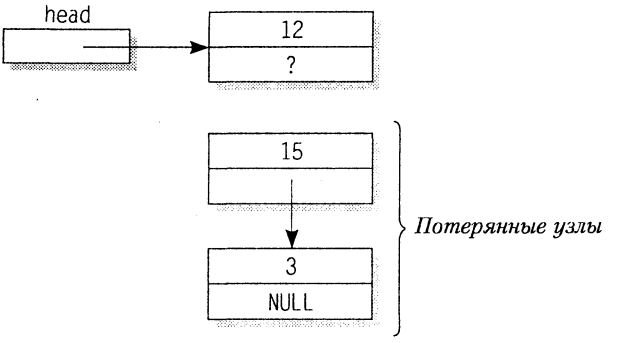


Рис. 15.4. Потерянные узлы

Эта ситуация, иногда называемая *утечкой памяти*, сковывает память, занятую потерянными узлами, на все время выполнения программы. Значительные утечки памяти могут привести к тому, что необходимая программе динамическая память будет исчерпана и выполнение программы завершится ошибкой. Более того, в некоторых системах утечка памяти в обычной пользовательской программе может вызвать сбой операционной системы. Во избежание таких проблем программа должна всегда содержать указатель на начало списка; как правило, таким указателем является переменная head.

Поиск в связном списке

В данном разделе мы разработаем функцию для поиска данных в связном списке. В ней используется тот же тип узла Node, что и в предыдущих разделах. (Определения типа Node и типа указателя приведены в листинге 15.1.) У нашей функции два аргумента: связный список и искомое целое число. Она возвращает указатель на первый узел, содержащий это число. Если заданное число не содержится ни в одном узле, функция возвращает NULL. Тогда программа сможет проверить, имеется ли число в списке, сравнив переменную-указатель со значением NULL. Объявление и пояснительный комментарий нашей функции таковы:

```
NodePtr search(NodePtr head, int target);
// Предусловие: переменная-указатель head ссылается на первый
// элемент связного списка. Переменная-указатель последнего
// узла содержит значение NULL. Если список пуст, переменная
// head содержит значение NULL.
// Возвращает указатель на первый узел, содержащий искомое число.
// Если ни один узел не содержит это число, функция возвращает NULL.
```

Для перемещения по списку в поисках числа (переменная target) мы воспользуемся локальной переменной-указателем here. Единственным способом перемещения по связному списку (или другой конструкции, составленной из узлов и указателей) является последовательный переход по указателям. Поэтому начнем с того, что присвоим локальной переменной here указатель на первый элемент списка, а затем будем переходить от узла к узлу, следуя содержащимся в них указателям. Эта технология показана на рис. 15.5. С пустыми списками связаны некоторые особенности, отвлекающие от сути алгоритма, но мы поговорим о них позднее, а пока будем считать, что список не является пустым. Итак, поиск элемента в связном списке выполняется в соответствии со следующим алгоритмом.

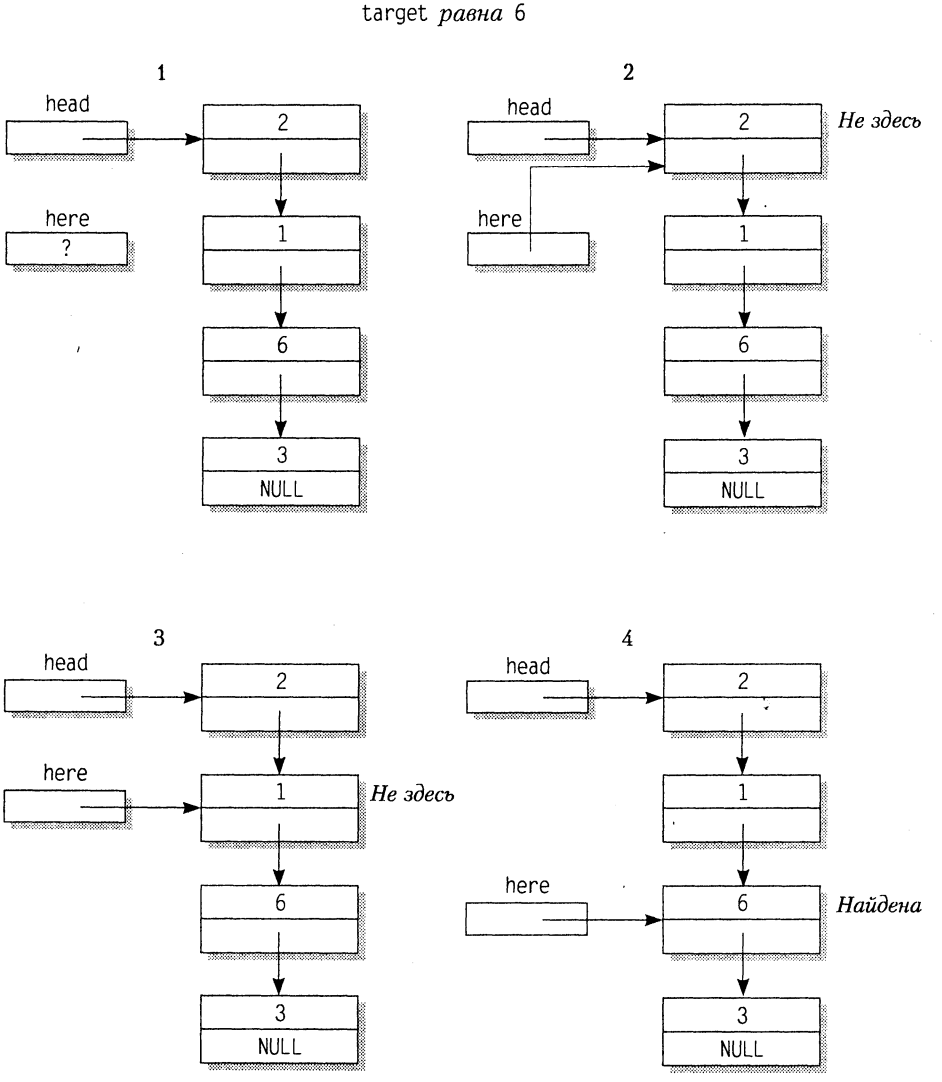


Рис. 15.5. Поиск в связном списке

Псевдокод для функции search следующий.

1. Присвоить локальной переменной here указатель на первый элемент связного списка.

```
while (here не указывает на узел, содержащий target
и here не указывает на последний узел)
{
    Присвоить here указатель на следующий узел списка.
}
if (узел, на который указывает here содержит значение переменной target)
    return here;
else
    return NULL;
```

2. Нам необходимо присвоить переменной here указатель на следующий узел — тот, на который указывает переменная-член link узла, на который в данный момент указывает переменная here. На эту переменную-член link можно сослаться с помощью выражения

```
here->link
```

Таким образом, чтобы переменная here указывала на следующий узел, ей должно быть присвоено значение приведенного выше указателя:

```
here = here->link;
```

Соединив все это вместе, получим следующую версию кода алгоритма поиска.

Предварительная версия кода функции search такова.

```
here = head;

while (here->data != target &&
                     here->link != NULL)

    here = here->link;

if (here->data == target)
    return here;
else
    return NULL;
```

Обратите внимание на логическое выражение в операторе while. В нем проверяется, указывает ли переменная here на последний узел, сравнивая значение переменной-члена here->link со значением NULL.

Теперь нужно вернуться назад и решить, что делать с пустым списком. Обратившись к приведенному выше коду, мы увидим, что с пустым списком он работает неправильно. Если список пуст, переменная here содержит значение NULL, и выражения

```
here->data
here->link
```

не определены.

Когда here содержит NULL, она не указывает ни на один узел, так что нечего говорить о членах data или link. Поэтому для пустого списка добавим в функцию отдельную ветвь кода. Полное определение функции search приведено в листинге 15.2.

Листинг 15.2. Функция для поиска узла в связном списке

Объявление функции

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

NodePtr search(NodePtr head, int target);
// Предусловие: переменная-указатель head указывает на первый
// элемент связного списка. Переменная-указатель последнего
// узла содержит значение NULL. Если список пуст, переменная
// head содержит значение NULL.
// Возвращает указатель на первый узел, содержащий искомое число.
// Если ни один узел не содержит это число, функция возвращает NULL.
```

Определение функции

```
// Используем библиотеку cstddef:
NodePtr search(NodePtr head, int target)
{
    NodePtr here = head;

    if (here == NULL)
    {
        return NULL;   // В случае пустого списка.
    }
    else
    {
        while (here->data != target &&
                                    here->link != NULL)
            here = here->link;

        if (here->data == target)
            return here;
        else
            return NULL;
    }
}
```

Указатели в качестве итераторов

Итератор — это конструкция, позволяющая перебирать элементы данных, хранящиеся в некоторой структуре, и выполнять над ними требуемые действия. Итератор может быть объектом какого-то класса или более простой структурой, например индексом массива. Удобным примером итератора является указатель. Основной принцип его использования легко продемонстрировать на примере связного

списка. Использование указателя в качестве итератора заключается в последовательном переходе от узла к узлу так, что вначале указатель-итератор ссылается на первый элемент списка, а в конце — на последний. Вот общая схема этого процесса:

```
тип_узла *iter;
for (iter = Head; iter != NULL; iter = iter->Link)
    Делайте с узлом, на который указывает iter, все, что хотите.
```

Здесь Head — указатель на начало списка, а Link — имя переменной-члена узла списка, указывающей на следующий узел.

Так, вывод данных всех узлов связного списка можно выполнить с помощью следующего кода:

```
NodePtr iter;   // Эквивалентно: Node *iter;
for (iter = head; iter != NULL; iter = iter->Link)
    cout << (iter->data);
```

Определения типов Node и NodePtr приведены в листинге 15.2.

Вставка и удаление элементов в середине списка

Теперь разработаем функцию для вставки узла в заданное место списка. Если узлы должны располагаться в определенном порядке, например по алфавиту, то нельзя вставлять все новые элементы в начало или в конец списка. Поэтому необходима функция, вставляющая элемент после заданного узла связного списка. Будем считать, что у нас имеется другая функция или часть программы, правильно присвоившая переменной after_me указатель на тот узел списка, после которого следует вставить новый узел, как показано на рис. 15.6.

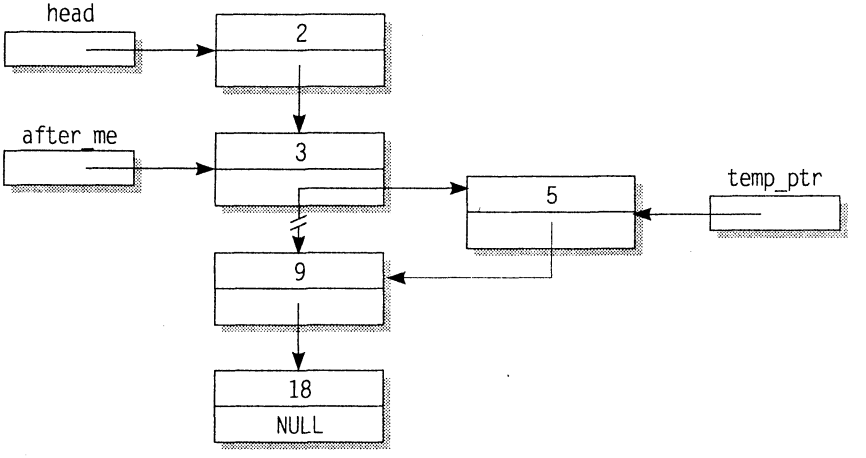


Рис. 15.6. Вставка нового узла в середину связного списка

Та же технология применима и к узлам с данными других типов, но для удобства мы воспользуемся тем же типом, что и в предыдущих разделах. Необходимые

определения типов приведены в листинге 15.2. Функция, которую нужно написать, объявляется так:

```
void insert(NodePtr after_me, int the_number);
// Предусловие: after_me указывает на один из узлов связного списка.
// Постусловие: после узла, на который указывает after_me,
// добавлен новый узел, содержащий значение the_number.
```

Новый узел создается точно так же, как в функции head_insert (см. листинг 15.1). Эти две функции различаются тем, что одна из них вставляет узел в начало списка, а вторая — после заданного узла. Алгоритм выполнения вставки показан на рис. 15.6 и на языке C++ выражается следующим кодом:

```
// Добавление в список указателя на новый узел:
temp_ptr->link = after_me->link;
// Добавление в новый узел указателя из списка:
after_me->link = temp_ptr;
```

Порядок этих двух операторов исключительно важен. В первом операторе присваивания требуется значение указателя after_me->link до его изменения. Полная функция приведена в листинге 15.3.

Листинг 15.3. Функция для добавления узла в середину связного списка

Объявление функции

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;

void insert(NodePtr after_me, int the_number);
// Предусловие: after_me указывает на один из узлов связного списка.
// Постусловие: после узла, на который указывает after_me,
// добавлен новый узел, содержащий значение the_number.
```

Определение функции

```
void insert(NodePtr after_me, int the_number)
{
    NodePtr temp_ptr;
    temp_Ptr = new Node;

    temp_ptr->data = the_number;

    temp_ptr->link = after_me->link;
    after_me->link = temp_ptr;
}
```

Проанализировав код функции insert, вы увидите, что она работает правильно, даже если узел, на который указывает after_me, является последним узлом списка. Но она не может вставить узел в начало связного списка. Для этого следует использовать функцию head_insert, приведенную в листинге 15.1.

Функция insert помогает поддерживать список отсортированным — по порядку числовых значений, по алфавиту или в каком-либо другом порядке. Для вставки

нового узла в нужное место достаточно изменить два указателя независимо от длины списка и места вставки нового узла. Если бы вместо связного списка использовался массив, то в самом худшем случае для освобождения места под новый элемент пришлось бы копировать весь массив. И несмотря на издержки, связанные с позиционированием указателя `after_me`, вставка в связный список обычно выполняется куда эффективнее, чем в массив.

Удалить узел из связного списка тоже очень просто. Этот процесс показан на рис. 15.7.

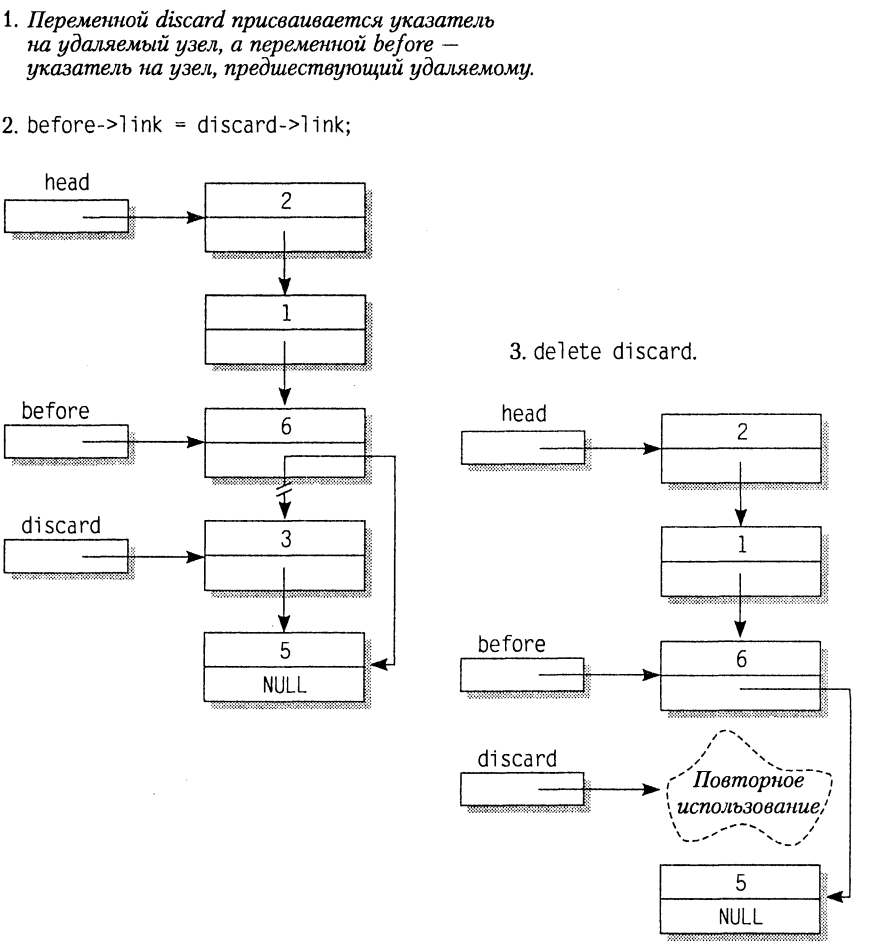


Рис. 15.7. Удаление узла из связного списка

После установки указателей `before` (на узел, предшествующий удаляемому) и `discard` (на удаляемый узел) для удаления узла достаточно выполнить следующий оператор:

```
before->link = discard->link;
```

Если удаленный из списка узел программе больше не нужен, его следует уничтожить и вернуть занимаемую им память в динамическую область. Это, как обычно, делается с помощью оператора delete:

```
delete discard;
```

Ловушка: использование оператора присваивания с динамическими структурами данных

Если head1 и head2 — переменные-указатели и head1 ссылается на первый узел связного списка, то следующий оператор присваивания:

```
head2 = head1;
```

помещает в переменную head2 указатель на этот же узел и, следовательно, на этот же связный список.

Однако при этом в программе остается один связный список, а не два. Когда что-либо изменяется в списке, на который указывает head1, то изменяется и список, на который указывает head2, поскольку это один и тот же список.

Если переменная head1 указывает на связный список и вы хотите, чтобы переменная head2 указывала на другую, идентичную копию этого списка, приведенный выше оператор присваивания не годится. Однако его можно перегрузить так, чтобы он создавал копию списка. О перегрузке оператора присваивания рассказывается в главе 12.

Упражнения для самопроверки

5. Напишите определения типов для узлов и указателей связного списка. Назовите тип узла NodeType, а тип указателя — PointerType. Связные списки будут списками букв.
6. Обычно связный список задается как указатель на его первый узел, но у пустого списка этот узел отсутствует. Какое значение указателя обычно используется для представления пустого списка?
7. Предположим, программа содержит следующие определения типов структуры и указателя, а также объявления переменных-указателей:

```
struct Node
{
    double data;
    Node *next;
};

typedef Node* Pointer;
Pointer p1, p2;
```

Допустим, что p1 указывает на узел связного списка. Напишите код, присваивающий переменной p1 указатель на следующий узел связного списка. (Указатель p2 предназначен для следующего упражнения и в данном упражнении не используется.)

8. Предположим, программа содержит определения и объявления, приведенные в упражнении 7. Допустим, что p2 указывает на узел связного списка, не являющийся последним. Напишите код, удаляющий узел, следующий за узлом, на который указывает p2. После его выполнения связный список должен остаться таким же, но без одного узла.
Подсказка. Возможно, потребуется объявить еще одну переменную-указатель.

9. Выберите ответ и поясните его.

Для большого массива и большого списка, содержащих объекты одного и того же типа, вставка нового объекта в указанную позицию связного списка по сравнению со вставкой объекта в массив выполняется:

а) более эффективно;

б) менее эффективно;

в) почти с той же эффективностью;

в) ответ зависит от размера списка.

15.2. Применение связных списков

Многие же будут первые последними и последние первыми.
Евангелие от Матфея, 19.30

Связные списки имеют множество применений. В этом разделе приводится лишь один из примеров – определение класса, в основе реализации которого лежит связный список.

Стеки

Стек – это структура данных, из которой они извлекаются в порядке, обратном их добавлению. Предположим, что мы поместили в стек букву 'A', затем 'B', а затем 'C'. Тогда извлечь их из стека можно будет только в следующем порядке: 'C', потом 'B', а потом 'A'. Этот принцип работы стека показан на рис. 15.8. Вы видите, что стек можно представить как ямку в земле, и чтобы достать из нее что-нибудь, нужно сначала вынуть верхние элементы. Поэтому стек иногда называют структурой данных типа «последним вошел – первым вышел».

Стеки используются для многих программных задач. В главе 13 было показано, как компьютерная система отслеживает с его помощью рекурсивные вызовы функции. Однако такое сложное приложение мы, конечно, разрабатывать не будем,

а создадим программу попроще. (Для понимания примера, приведенного в этом разделе, можно не читать главу 13.)

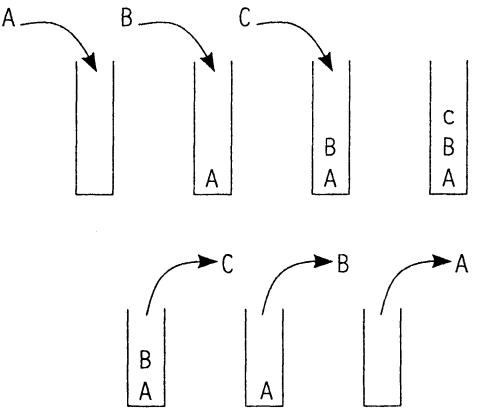


Рис. 15.8. Стек

Пример: класс Stack

Интерфейс нашего класса Stack, реализующего стек на основе связного списка, приведен в листинге 15.4. Этот стек используется для хранения данных типа char. Подобный стек можно определить для хранения данных любого другого типа. Над стеком можно выполнять две базовые операции: добавление элемента в стек и удаление их из него. Добавление элемента в стек называется *проталкиванием*, а удаление — *выталкиванием*. Соответствующие функции-члены класса Stack мы назовем push и pop.

В листинге 15.5 показана простая программа, в которой используется класс Stack. Она считывает слово по одной букве и помещает буквы в стек, а затем по одной извлекает буквы из стека и выводит их на экран. Поскольку данные удаляются из стека в порядке, обратном их добавлению, буквы, составляющие слово, оказываются выведены в обратном порядке.

Из листинга 15.6 видно, что наш класс реализован на основе связного списка, первый элемент которого представляет вершину стека. Переменная-член top является указателем на первый элемент списка.

Листинг 15.4. Файл интерфейса класса Stack

```
// Это заголовочный файл stack.h.
// В нем содержится интерфейс класса Stack,
// представляющего стек символов.
#ifndef STACK_H
#define STACK_H
namespace stacksavitch
{
    struct StackFrame
    {
        char data;
```

```
        StackFrame *link;
    };
    typedef StackFrame* StackFramePtr;

    class Stack
    {
    public:
        Stack();
        // Инициализирует объект пустым стеком.

        Stack(const Stack& a_stack);
        // Конструктор копирования.

        ~Stack();
        // Уничтожает стек и возвращает всю занимаемую им память
        // в динамическую область.

        void push(char the_symbol);
        // Постусловие: символ, определяемый значением аргумента the_symbol,
        // добавлен в стек.

        char pop();
        // Предусловие: стек не пуст.
        // Возвращает верхний символ стека
        // и удаляет его из стека.

        bool empty() const;
        // Возвращает true, если стек пуст.
        // В противном случае возвращает false.
    private:
        StackFramePtr top;
    };
}   // stacksavitch

#endif   // STACK_H
```

Листинг 15.5. Программа, в которой используется класс Stack

```
// Программа, демонстрирующая использование класса Stack.
#include <iostream>
#include "stack.h"
using namespace std;
using namespace stacksavitch;

int main()
{
    Stack s;
    char next, ans;

    do
    {
        cout << "Enter a word: ";
        cin.get(next);
        while (next != '\n')
        {
            s.push(next);
            cin.get(next);
        }
```

продолжение

Листинг 15.5 (*продолжение*)

```
        cout << "Written backward that is: ";
        while ( ! s.empty() )
            cout << s.pop();
        cout << endl;

        cout << "Again?(y/n): ";
        cin >> ans;
        cin.ignore(10000, '\n');
    }while (ans != 'n' && ans != 'N');

    return 0;
}
```

Пример диалога

```
Enter a word: straw
Written backward that is: warts
Again?(y/n): y
Enter a word: C++
Written backward that is: ++C
Again?(y/n): n
```

Листинг 15.6. Реализация класса Stack

```
// Это файл реализации stack.cpp.
// Он содержит реализацию класса Stack.
// Интерфейс класса Stack содержится в заголовочном файле stack.h.
#include <iostream>
#include <cstddef>
#include "stack.h"
using namespace std;

namespace stacksavitch
{
    // Используем библиотеку cstddef:
    Stack::Stack() : top(NULL)
    {
        // Тело функции намеренно оставлено пустым.
    }

    // Используем библиотеку cstddef:
    Stack::Stack(const Stack& a_stack)
    ... Определение конструктора копирования вам предлагается написать в упражнении 11. ...

Stack::~Stack()
    {
        char next;
        while (! empty())
           next = pop();   // Функция pop вызывает оператор delete.
    }

    // Используем библиотеку cstddef:
    bool Stack::empty() const
    {
        return (top == NULL);
    }
```

```
    // Используем библиотеку cstddef:
    void Stack::push(char the_symbol)
    ... Определение функции-члена push вам предлагается написать в упражнении 10. ...

    // Используем библиотеку классов iostream:
    char Stack::pop()
    {
        if (empty())
        {
            cout << "Error: popping an empty stack.\n";
            exit(1);
        }

        char result = top->data;

        StackFramePtr temp_ptr;
        temp_ptr = top;
        top = top->link;

        delete temp_ptr;

        return result;
    }
}   // stacksavitch
```

Определение функции-члена push вам предлагается написать в упражнении 10. Однако алгоритм этой задачи мы уже рассмотрели. Код функции-члена push отличается от кода функции head_insert, приведенного в листинге 15.1, только тем, что в функции push указатель назван не head, а top.

Пустой стек — это просто пустой связный список, поэтому он реализован как переменная-указатель top, содержащая значение NULL. Так что реализация используемого по умолчанию конструктора и функции-члена empty очевидны.

Определение конструктора копии немного сложнее, но используемые в нем технологии вы хорошо знаете, поэтому в упражнении 11 вам предлагается разработать этот конструктор самостоятельно.

Функция-член pop сначала проверяет, не пуст ли стек. Если это так, она удаляет из стека верхний элемент и присваивает его значение локальной переменной result следующим образом:

```
char result = top->data;
```

После сохранения символа из верхнего элемента стека в переменной result указатель top перемещается к следующему узлу связного списка, а верхний элемент стека удаляется. Перемещение указателя выполняется так:

```
top = top->link;
```

Однако перед перемещением указателя адрес удаляемого элемента стека присваивается временной переменной-указателю temp_ptr, чтобы этот элемент можно было удалить с помощью приведенного ниже вызова delete:

```
delete temp_ptr;
```

Каждый узел, удаляемый из связного списка с помощью функции-члена pop, обязательно уничтожается посредством оператора delete. Поэтому единственной задачей деструктора является удаление из стека всех имеющихся в нем элементов с помощью последовательных вызовов функции pop. В результате занимаемая память возвращается в динамическую область.

Упражнения для самопроверки

10. Приведите определение функции-члена push класса Stack, определенного в листинге 15.4.
11. Приведите определение конструктора копирования класса Stack, определенного в листинге 15.4.

Резюме

- Узел — это структура (struct) или объект класса с одной или несколькими переменными-членами и указателем. Такие узлы можно соединять с помощью входящих в них указателей для формирования структур данных, размер которых может увеличиваться и уменьшаться во время выполнения программы.
- Связный список — это список узлов, каждый из которых содержит указатель на следующий узел списка.
- Конец связного списка (или другой структуры данных, элементы которой связаны с помощью указателей) отмечается установкой указателя последнего узла в значение NULL.
- Стек является структурой данных типа «последним вошел — первым вышел». Он может быть реализован в виде связного списка.

Ответы к упражнениям для самопроверки

1. ```
   Sally
   Sally
   18
   18
   ```

   Обратите внимание, что выражения (*head).name и head->name, а также выражения (*head).number и head->number означают одно и то же.

2. Наилучший ответ:

   ```
   head->next = NULL;
   ```

   Однако ответ

   ```
   (*head).next = NULL;
   ```

   тоже является правильным.

3. ```
   delete head;
   ```

4. ```
   head->item = "Wilbur's brother Orville";
   ```
   ```

5.
```
struct NodeType
{
    char data;
    NodeType *link;
};
typedef NodeType* PointerType;
```

6. Признаком конца списка служит значение указателя NULL.

7.
```
p1 = p1-> next;
```

8.
```
Pointer discard;
discard = p2->next;
// Теперь discard указывает на удаляемый узел.
p2->next = discard->next;
```

   Этого достаточно для удаления узла из связного списка. Однако если этот узел больше не будет использоваться программой, его следует уничтожить с помощью оператора delete, вот так:

```
delete discard;
```

9. Ответ а). Вставка нового элемента в заданное место большого связного списка более эффективна, чем вставка его в большой массив. Для вставки элемента в список независимо от размера списка всегда требуется пять операций, большинство из которых — присваивание указателей. Для вставки элемента в массив в среднем приходится перемещать половину массива.

   Однако для маленьких списков справедлив и ответ в).

10.
```
// Используем библиотеку cstddef:
void Stack::push(char the_symbol)
{
    StackFramePtr temp_ptr;
    temp_ptr = new StackFrame;
    temp_ptr->data = the_symbol;
    temp_ptr->link = top;
    top = temp_ptr;
}
```

11.
```
// Используем библиотеку cstddef:
Stack::Stack(const Stack& a_stack)
{
    if (a_stack.top == NULL)
        top = NULL;
    else
    {
        StackFramePtr temp = a_stack.top;   // Указатель temp перемещается
                                            // по узлам стека a_stack сверху вниз.
        StackFramePtr end;   // Указывает на конец нового стека.

        end = new StackFrame;
        end->data = temp->data;
        top = end;
        // Первый узел создается и заполняется данными.
        // Затем ПОСЛЕ него добавляются новые узлы.

        temp = temp->link;
```

```
            while (temp != NULL)
            {
                end->link = new StackFrame;
                end = end->link;
                end->data = temp->data;
                temp = temp->link;
            }
            end->link = NULL;
        }
    }
```

Практические задания

1. Напишите функцию типа void, принимающую список целых чисел и заменяющую порядок его узлов обратным. У нее будет один передаваемый по ссылке параметр — указатель на первый элемент списка. После вызова этой функции он будет указывать на начало связного списка с теми же узлами, что и у исходного, но расположенными в обратном порядке. Обратите внимание, что ваша функция не должна ни создавать узлы, ни уничтожать их — их нужно просто переупорядочить. Включите функцию в подходящую отладочную программу.
2. Напишите функцию merge_lists с двумя передаваемыми по ссылке аргументами-указателями, указывающими на начало двух связных списков значений типа int. Предполагается, что эти списки отсортированы таким образом, что первый узел содержит наименьшее число, второй — следующее наименьшее из оставшихся чисел и т. д. Функция возвращает указатель на первый узел нового связного списка, содержащего все узлы двух исходных списков, отсортированных по возрастанию значений. Обратите внимание, что ваша функция не должна ни создавать узлы, ни уничтожать их. После ее вызова обе переменные-аргументы должны содержать значение NULL.
3. Очередь — это структура данных, похожая на стек, но действующая противоположным образом. Если стек функционирует по принципу «последним вошел — первым вышел», то очередь функционирует по принципу «первым вошел — первым вышел». Например, если добавить в очередь значений типа char символ 'A', затем 'B' и затем 'C', они будут удаляться в том же порядке: 'A', затем 'B' и затем 'C'. Напишите класс для очереди символов с интерфейсом, подобным интерфейсу класса Stack, приведенному в листинге 15.4. Функция-член для добавления символа будет называться add, а не push, а функция-член для удаления символа — remove, а не pop. Напишите полную реализацию всех функций-членов и перегруженных операторов класса, а также программу для отладки класса.
4. Разработайте и реализуйте класс, объекты которого представляют многочлены. Многочлен

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$$

должен быть реализован как связный список. Каждый его узел будет содержать значение типа int для степени x и значение типа int для соответствующего коэффициента. Класс должен поддерживать операции сложения, вычитания, умножения и вычисления значения многочлена. Перегрузите операторы +, - и * для сложения, вычитания и умножения.

Вычисление значения многочлена должно выполняться функцией-членом с одним аргументом типа int. Эта функция возвращает значение, полученное путем подстановки значения аргумента в переменную x и выполнения указанных операций. У класса должно быть четыре конструктора: используемый по умолчанию, конструктор копирования, конструктор с единственным аргументом типа int для создания многочлена с единственным константным членом, равным значению аргумента, и конструктор с двумя аргументами типа int для создания многочлена с единственным членом, коэффициент и степень которого заданы двумя аргументами. (Согласно приведенной выше формуле конструктор с одним аргументом создает многочлен a_0, а конструктор с двумя аргументами создает многочлен $a_n x^n$.) Напишите подходящий деструктор. Добавьте функции-члены для ввода и вывода многочленов.

Чтобы задать многочлен, пользователь вводит следующее:

$a_n x\hat{}n + a_{n-1} x\hat{}n-1 + \ldots + a_0$

Однако если коэффициент a_i равен нулю, пользователь может пропустить член $a_i x\hat{}i$. Например, многочлен

$3x^4 + 7x^2 + 5$

можно ввести в виде:

```
3x^4 + 7x^2 + 5
```

Если коэффициент отрицателен, вместо знака плюс используется знак минус, как в следующих примерах:

```
3x^5 - 7x^3 + 2x^1 - 8
-7x^4 + 5x^2 + 9
```

Знак минус перед многочленом, как во втором из двух приведенных примеров, применим только к первому коэффициенту; он не меняет знак всего многочлена. Многочлены выводятся в том же формате. Члены с нулевыми коэффициентами не выводятся.

Для упрощения ввода можно полагать, что каждый многочлен вводится в отдельной строке и в нем всегда имеется константный член a_0. Если он отсутствует, пользователь просто вводит 0, как в следующем примере:

```
12x^8 + 3x^2 + 0
```

5. В этом задании нужно переделать практическое задание 10 из главы 10 с использованием связного списка вместо массива. Этот список будет содержать значения типа double. Придется изменить некоторые функции-члены: конструктор, используемый по умолчанию; функцию-член add_item, добавляющую в список заданное значение типа double; функцию-член full(), проверяющую, полон ли список; дружественную функцию, перегружающую оператор <<.

6. Более сложная версия задания 5 заключается в следующем. Нужно написать класс List, такой же, как в задании 5, со следующими функциями-членами:
   - List() — используемый по умолчанию конструктор;
   - double List::front() — возвращает первый элемент списка;
   - double List::back() — возвращает последний элемент списка;
   - double List::current() — возвращает «текущий» элемент;
   - void List::advance() — назначает «текущим» следующий элемент списка;
   - void List::reset() — назначает «текущим» первый элемент списка;
   - void List::insert(double after_me, double insert_me) — вставляет значение insert_me в список после значения after_me и увеличивает на единицу значение закрытой переменной-члена count;
   - int size() — возвращает количество элементов в списке;
   - friend istream& operator<< (istream& ins, double write_me).

   Среди закрытых членов класса должны быть такие:

```
node* head;
node* current;
int count;
```

   и, возможно, еще один указатель.

   Для узлов связного списка потребуется следующая структура (объявленная вне класса):

```
struct node
{
    double item;
    node *next;
};
```

   При создании проектов любого размера важно следовать инкрементальной стратегии разработки, и этот проект — не исключение. Напишите определение класса List, но не реализуйте его члены. Поместите определение этого класса в файл list.h. Затем включите этот файл с помощью директивы #include "list.h" в файл, содержащий функцию int main(){}, и откомпилируйте его. В результате вы сразу выявите синтаксические ошибки и многие опечатки, которые труднее будет найти и исправить после написания членов класса. Затем по очереди пишите и компилируйте члены класса, пока их не окажется достаточно для написания отладочного кода функции main.

7. Это более сложная версия проекта 6 с использованием шаблонов (о которых рассказывалось в главе 14). Напишите определение шаблона класса для списка объектов, тип которых задается как параметр типа. Полностью реализуйте все функции-члены и перегруженные операторы. Протестируйте шаблон для следующего типа структуры:

```
struct Person
{
    string name;
    string phone_number;
};
```

Глава 16 Наследование и полиморфизм

Когда есть все возможности и средства.

Уильям Шекспир

В настоящее время все шире используется мощная технология объектно-ориентированного программирования, которое вобрало в себя лучшие идеи структурного программирования и объединило их с несколькими новыми концепциями, в результате чего появился особый способ организации программ. Язык C++ полностью поддерживает базовые принципы ООП – инкапсуляцию, полиморфизм и наследование. Первый из них мы уже рассматривали в главе 6 этой книги, а сейчас поговорим подробно о наследовании и полиморфизме.

16.1. Наследование

Если мы хотим что-то изменить в ребенке, нужно сначала подумать, не лучше ли изменить это в себе.

Карл Густав Юнг

Одной из наиболее эффективных возможностей языка C++ является наследование, позволяющее создавать одни классы на базе других. *Наследование* – это процесс создания нового класса, называемого *производным*, на основе другого, именуемого *базовым классом*. Производный класс автоматически включает все переменные-члены и функции-члены базового класса и, кроме того, может содержать дополнительные функции и переменные.

В главе 5 говорилось, что выражение «класс D является производным от класса B» означает следующее: класс D содержит все элементы класса B и некоторые дополнительные компоненты. Когда класс D произведен от класса B, то класс B называют базовым, а D – производным. Класс D также называют *дочерним классом*, а B – *родительским*.

В качестве примера рассматривался стандартный класс ifstream, производный от класса istream путем добавления функций-членов (таких, как open и close). При этом мы отмечали, что поток cin относится к классу всех входных потоков (istream), но не относится к классу файловых входных потоков (ifstream), поскольку не содержит функций-членов open и close, определенных в производном классе ifstream.

Производные классы

Предположим, мы создаем программу, чтобы работать с записями о двух категориях сотрудников: получающих фиксированную ставку и тех, чей труд оплачивается почасово. Для группировки этих записей и соответствующих им классов существует определенная иерархия, на вершине которой располагается класс, соответствующий множеству всех сотрудников.

Это множество делится на два подмножества: в первое из них входят сотрудники, чья работа оплачивается почасово, а во второе – сотрудники, получающие фиксированный оклад ежемесячно или еженедельно. В программе может и не потребоваться класс всех сотрудников, однако полезно проанализировать те данные, которые являются универсальными для каждого работника и не зависят от вида оплаты труда. Например, все сотрудники имеют фамилии и идентификационные номера, поэтому для каждого из них можно определить функции-члены, устанавливающие и изменяющие эти значения вне зависимости от системы оплаты труда конкретного работника.

Средствами языка C++ можно определить класс Employee, представляющий всех сотрудников (как тех, кто получает почасовую оплату, так и работающих на ставке), и использовать этот класс в качестве базового для определения классов, представляющих две названные категории сотрудников. В листингах 16.1 и 16.2 показано, как это можно сделать.

Листинг 16.1. Интерфейс базового класса Employee

```
// Это заголовочный файл employee.h.
// В нем содержится интерфейс класса Employee, который будет
// использоваться в качестве базового для определения
// классов, соответствующих различным категориям сотрудников.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

namespace employeessavitch
{

    class Employee
    {
    public:
        Employee();
        Employee(string the_name, string the_ssn);
        string get_name() const;
        string get_ssn() const;
        double get_net_pay() const;
```

```
        void set_name(string new_name);
        void set_ssn(string new_ssn);
        void set_net_pay(double new_net_pay);
        void print_check() const;
    private:
        string name;
        string ssn;
        double net_pay;
    };

}   // employeessavitch

#endif   // EMPLOYEE_H
```

Листинг 16.2. Реализация базового класса Employee

```
// Это файл реализации : employee.cpp.
// Он содержит реализацию класса Employee.
// Интерфейс класса Employee находится в заголовочном файле employee.h.
#include <string>
#include <cstdlib>
#include <iostream>
#include "employee.h"
using namespace std;

namespace employeessavitch
{
    Employee::Employee() : name("No name yet"), ssn("No number yet"), net_pay(0)
    {
        // Тело намеренно оставлено пустым.
    }

    Employee::Employee(string the_name, string the_number)
       : name(the_name), ssn(the_number), net_pay(0)
    {
        // тело намеренно оставлено пустым
    }

    string Employee::get_name() const
    {
        return name;
    }

    string Employee::get_ssn() const
    {
        return ssn;
    }

    double Employee::get_net_pay() const
    {
        return net_pay;
    }

   void Employee::set_name(string new_name)
    {
        name = new_name;
    }
    void Employee::set_ssn(string new_ssn)
```

продолжение

Листинг 16.2 (*продолжение*)

```
    {
        ssn = new_ssn;
    }

    void Employee::set_net_pay (double new_net_pay)
    {
        net_pay = new_net_pay;
    }

    void Employee::print_check() const
    {
        cout << "\nERROR: print_check FUNCTION CALLED FOR AN \n"
             << "UNDIFFERENTIATED EMPLOYEE. Aborting the program.\n"
             << "Check with the author of the program about this bug.\n";
        exit(1);
    }

}   // employeessavitch
```

В этой программе можно, конечно, ограничиться использованием только класса Employee, но наша цель – определить его как базовый, от которого затем можно производить специализированные классы, соответствующие разным категориям сотрудников. В частности, определение функции print_check в производных классах нужно будет постоянно изменять, чтобы сотрудники различных категорий могли получать чеки на разные денежные суммы. Это отражено в определении функции класса Employee листинга 16.2. Понятно, что выводить чек для сотрудника неопределенной категории не имеет смысла, поскольку способ оплаты его труда не известен. Так что данная функция, вызванная для объекта базового класса Employee, просто останавливает работу программы с сообщением об ошибке.

В класс, производный от Employee, автоматически включаются все его переменные-члены (name, ssn и net_pay), а также все его функции-члены, в том числе print_check, get_name, set_name и др., перечисленные в листинге 16.1. То есть производный класс наследует переменные-члены и функции-члены базового класса.

Файлы интерфейса классов HourlyEmployee и SalariedEmployee, производных от класса Employee, приведены в листингах 16.3 и 16.4. Хотя C++ этого и не требует, данный родительский и два его дочерних класса, поскольку они связаны между собой, помещены в общее пространство имен.

Сначала рассмотрим класс HourlyEmployee.

Листинг 16.3. Интерфейс производного класса HourlyEmployee

```
// Это заголовочный файл hourlyemployee.h.
// В нем содержится интерфейс класса HourlyEmployee.
#ifndef HOURLYEMPLOYEE_H
#define HOURLYEMPLOYEE_H

#include <string>
#include "employee.h"

using namespace std;
```

```
namespace employeessavitch
{

    class HourlyEmployee : public Employee
    {
    public:
        HourlyEmployee();
        HourlyEmployee(string the_name, string the_ssn,
                       double the_wage_rate, double the_hours);
        void set_rate(double new_wage_rate);
        double get_rate() const;
        void set_hours(double hours_worked);
        double get_hours() const;
        void print_check();   // Для того чтобы переопределить унаследованную функцию,
                              // достаточно еще раз привести ее объявление.
    private:
        double wage_rate;
        double hours;
    };

}   // employeessavitch

#endif   // HOURLYMPLOYEE_H
```

Листинг. 16.4. Интерфейс производного класса SalariedEmployee

```
// Это заголовочный файл salariedemployee.h.
// В нем содержится интерфейс класса SalariedEmployee.
#ifndef SALARIEDEMPLOYEE_H
#define SALARIEDEMPLOYEE_H

#include <string>
#include "employee.h"

using namespace std;

namespace employeessavitch
{

    class SalariedEmployee : public Employee
    {
    public:
        SalariedEmployee();
        SalariedEmployee (string the_name, string the_ssn,
                          double the_weekly_salary);
        double get_salary() const;
        void set_salary(double new_salary);
        void print_check();
    private:
        double salary;   // Недельный оклад.
    };

}   // employeessavitch

#endif   // SALARIEDEMPLOYEE_H
```

Обратите внимание, что определение производного класса начинается так же, как определение любого другого класса, но за его именем следуют двоеточие, зарезервированное слово public и имя базового класса:

```
class HourlyEmployee : public Employee
{
```

Производный класс (в нашем примере HourlyEmployee) автоматически наследует все переменные-члены и функции-члены базового класса (здесь Employee) и может содержать дополнительные переменные-члены и функции-члены.

Заметьте, что в определении класса HourlyEmployee объявлены две дополнительные переменные, wage_rate и hours, и отсутствуют переменные-члены name, ssn и net_pay, но, тем не менее, они есть у каждого объекта данного класса, поскольку он наследует их от класса Employee. То есть определение производного класса (такого, как HourlyEmployee) содержит только его собственные переменные-члены, а унаследованные от базового класса переменные-члены в нем не упоминаются — они включаются в производный класс автоматически. Таким образом, каждый объект класса HourlyEmployee имеет пять переменных-членов: name, ssn, net_pay, wage_rate и hours.

Класс HourlyEmployee наследует от класса Employee не только все переменные-члены, но и все функции-члены — get_name, get_ssn, get_net_pay, set_name, set_ssn, set_net_pay и print_check.

Помимо унаследованных в производный класс включаются новые переменные-члены и функции-члены. Их объявления содержатся в определении этого класса. Например, в производный класс HourlyEmployee добавлены переменные-члены wage_rate и hours, а также новые функции-члены, названные set_rate, get_rate, set_hours и get_hours. Их объявления приведены в листинге 16.3. Обратите внимание, что в определение производного класса включаются объявления только тех унаследованных функций-членов, которые переопределяются в производном классе. Они добавляются в данное определение подобно объявлению функции print_check из базового класса Employee. На определении конструктора производного класса здесь мы пока останавливаться не будем, эта тема рассмотрена в следующем разделе.

Унаследованные члены

В производный класс автоматически включаются все переменные-члены и обычные функции-члены базового класса. Такие члены базового класса называют *унаследованными*. Унаследованные функции-члены и переменные-члены приводятся в определении производного класса только в том случае, если программист хочет их переопределить. (Ниже в данной главе будет рассказано о том, что существуют некоторые специализированные функции-члены, например конструкторы, которые автоматически не наследуются.)

Файл реализации производного класса, такого как класс HourlyEmployee из листинга 16.5, содержит определения всех добавленных функций-членов. Заметьте, что определения унаследованных функций-членов не приводятся, если только

определение какой-нибудь функции не изменено в производном классе. Последний случай мы рассмотрим отдельно.

Листинг 16.5. Реализация производного класса HourlyEmployee

```
// Это файл реализации: hourlyemployee.cpp
// Он содержит реализацию класса HourlyEmployee.
// Интерфейс класса HourlyEmployee содержится
// в заголовочном файле hourlyemployee.h.
#include <string>
#include <iostream>
#include "hourlyemployee.h"
using namespace std;

namespace employeessavitch
{

    HourlyEmployee::HourlyEmployee() : Employee(), wage_rate(0), hours(0)
    {
        // Тело намеренно оставлено пустым.
    }

    HourlyEmployee::HourlyEmployee(string the_name, string the_number,
                                   double the_wage_rate, double the_hours)
    : Employee(the_name, the_number), wage_rate(the_wage_rate), hours(the_hours)
    {
        // Тело намеренно оставлено пустым.
    }

    void HourlyEmployee::set_rate(double new_wage_rate)
    {
        wage_rate = new_wage_rate;
    }

    double HourlyEmployee::get_rate() const
    {
        return wage_rate;
    }


    void HourlyEmployee::set_hours(double hours_worked)
    {
        hours = hours_worked;
    }

    double HourlyEmployee::get_hours() const
    {
        return hours;
    }

/* Мы вызывали функцию set_net_pay из функции print_check, поскольку она больше нигде не
используется. Это вопрос, скорее, из области бухгалтерии, чем программирования. Однако
заметьте, что C++ позволяет в производном классе удалить из определения функции print_check
модификатор const. */

    void HourlyEmployee::print_check()
    {
        set_net_pay(hours * wage_rate);
```

продолжение ⇩

Листинг 16.5 (*продолжение*)

```
        cout << "\n_____\n";
        cout << "Pay to the order of " << get_name() << endl;
        cout << "The sum of " << get_net_pay() << " Dollars\n";
        cout << "_____\n";
        cout << "Check Stub: NOT NEGOTIABLE\n";
        cout << "Employee Number: " << get_ssn() << endl;
        cout << "Hourly Employee. \nHours worked: " << hours
             << " Rate: " << wage_rate << " Pay: " << get_net_pay() << endl;
        cout << "_____\n";
    }

}    // employeessavitch
```

Родительский и дочерний классы

При описании производных классов часто используют ту же терминологию, что и для родственных связей, то есть базовый класс называют *родительским*, а производный — *дочерним*. Наследование естественно вписывается в эту терминологию, например, можно сказать, что дочерний класс наследует переменные-члены и переменные-функции от своего родительского класса. Часто аналогию с родственными связями развивают еще дальше, и родительский класс именуют *классом-предком*, а производный — *потомком*.

Определение унаследованной функции-члена в производном классе можно изменить, для того чтобы ее назначение в дочернем классе отличалось от аналогичного в родительском. Такое изменение называется *переопределением* унаследованной функции-члена. В нашем примере функция-член `print_check` переопределяется в производном классе `HourlyEmployee`, для чего в определение класса включается ее новое объявление и пишется ее новое определение — как будто создается новая функция-член, добавляемая в производный класс. Все это показано на примере переопределенной функции `print_check` класса `HourlyEmployee` (см. листинги 16.3 и 16.5).

Еще одним примером класса, производного от класса `Employee`, является `SalariedEmployee`. Его интерфейс приведен в листинге 16.4. Любой объект типа `SalariedEmployee` содержит все функции-члены, переменные-члены класса `Employee` и новые члены, указанные в определении класса `SalariedEmployee`. При этом в определении этого класса нет ни одной переменной-члена и ни одной функции-члена класса `Employee` за исключением функции `print_check`, определение которой изменено. Тем не менее класс `SalariedEmployee` содержит три унаследованные переменные: `name`, `ssn` и `net_pay`, а также собственную переменную `salary`. Обратите внимание, что переменные-члены и функции-члены класса `Employee` вновь объявлять не требуется — класс `SalariedEmployee` и так их содержит, он наследует данные переменные автоматически без всякого вмешательства программиста.

В классе `Employee` находится весь код, общий для классов `HourlyEmployee` и `SalariedEmployee`, что избавляет программиста от необходимости дважды программировать одно и то же. Применение технологии наследования позволяет повторно использовать код класса `Employee`.

Объект производного класса имеет более одного типа

В реальной жизни любой сотрудник, труд которого оплачивается почасово, является членом общего коллектива сотрудников. Точно так же обстоит дело с объектами C++. Поскольку класс HourlyEmployee является производным от класса Employee, каждый его объект может использоваться везде, где допускается использование объекта класса Employee. В частности, его можно применять в качестве аргумента функции, параметр которой определен как Employee. Объект класса HourlyEmployee можно присвоить переменной типа Employee. (Но учтите, что сделать наоборот, то есть присвоить объект Employee переменной типа HourlyEmployee, нельзя, поскольку работа не каждого сотрудника оплачивается почасово, и следовательно, представляющий его объект не обязательно принадлежит классу HourlyEmployee.) Конечно, это же верно и для любых других базовых и производных классов. Объект производного класса можно использовать везде, где допускается использование объекта базового класса.

Область действия этой концепции даже шире — она распространяется не только на дочерние классы, но и на классы, являющиеся производными от дочерних. Объект типа класса может использоваться везде, где допускается использование объекта его класса-предка. Если класс Child является производным от класса Ancestor, а класс Grandchild является производным от класса Child, то объект класса Grandchild может использоваться везде, где допускается использование объекта класса Ancestor.

Конструкторы в производных классах

Конструктор базового класса не наследуется производным классом, но его можно вызвать из определения конструктора производного класса, что обычно и делают. Поскольку конструктор базового класса инициализирует все переменные-члены, унаследованные производным классом от базового, конструктор производного класса обычно начинает свою работу с вызова конструктора базового класса.

Для этого используется специальный синтаксис, показанный в листинге 16.5. Ниже мы повторяем одно из определений конструкторов класса HourlyEmployee, приведенное в этом листинге:

```
HourlyEmployee::HourlyEmployee(string the_name,
       string the_number,  double the_wage_rate,
       double the_hours)
       : Employee(the_name, the_number),
                  wage_rate(the_wage_rate), hours(the_hours)
{
   // Тело намеренно оставлено пустым.
}
```

Часть кода после двоеточия — это раздел инициализации определения конструктора HourlyEmployee::HourlyEmployee. Строка кода Employee(the_name, the_number) представляет собой вызов конструктора базового класса Employee с двумя аргументами, the_name и the_number. Обратите внимание, что синтаксис вызова конструктора базового класса аналогичен синтаксису установки переменных-членов: выражение wage_rate(the_wage_rate) задает значение переменной wage_rate равным the_wage_rate. Поскольку вся работа выполняется в разделе инициализации, тело конструктора остается пустым.

Вот еще один конструктор класса HourlyEmployee из файла, приведенного в листинге 16.5:

```
HourlyEmployee::HourlyEmployee() : Employee(), wage_rate(0), hours(0)
{
    // Тело намеренно оставлено пустым.
}
```

В этом конструкторе для инициализации унаследованных переменных-членов вызывается определение используемого по умолчанию конструктора базового класса. Всегда включайте вызов одного из конструкторов базового класса в инициализационный раздел определения конструктора производного класса.

Если в таком определении не содержится вызов конструктора базового класса, то автоматически вызывается используемая по умолчанию версия конструктора базового класса без аргументов. Поэтому следующее определение используемого по умолчанию конструктора класса HourlyEmployee (без вызова Employee()):

```
HourlyEmployee::HourlyEmployee() : wage_rate(0), hours(0)
{
    // Тело намеренно оставлено пустым.
}
```

эквивалентно приведенной выше версии.

Но мы все же рекомендуем не полагаться на автоматический вызов, а всегда явно вызывать конструктор базового класса.

Объект производного класса содержит все переменные-члены базового класса. При вызове конструктора производного класса для них должна быть выделена память, и они должны быть инициализированы. Поскольку память выделяется конструктором базового класса, удобнее всего помещать инициализацию переменных именно в него. Вот почему в определение конструктора производного класса следует всегда включать вызов одного из конструкторов базового класса. Если же этого не сделать (в инициализационном разделе определения конструктора производного класса), то будет автоматически вызван используемый по умолчанию конструктор базового класса. (В случае отсутствия конструктора у базового класса, используемого по умолчанию, будет выведено сообщение об ошибке.)

Вызов конструктора базового класса является первым действием, выполняемым конструктором производного класса. Если класс B является производным от класса A и класс C является производным от класса B, сначала вызывается конструктор класса A, затем конструктор класса B, а потом выполняются оставшиеся операции конструктора класса C.

Конструкторы производных классов

Производный класс не наследует конструкторы базового класса. Но при определении конструктора производного класса можно и нужно вызвать конструктор базового класса (в инициализационном разделе определения конструктора производного класса).

Если не включить этот вызов, C++ автоматически вызовет используемый по умолчанию конструктор базового класса.

Ловушка: использование закрытых переменных-членов базового класса

Объект класса HourlyEmployee (см. листинги 16.3 и 16.5) наследует от класса Employee (см. листинги 16.1 и 16.2) переменную-член name. Так, следующий код

```
HourlyEmployee joe("Josephine", "123-45-6789", 0, 0);
```

установит значение данной переменной объекта joe равным "Josephine". (Кроме того, этот код присваивает переменной-члену ssn значение "123-45-6789", а переменным-членам wage_rate и hours значение 0.)

Для того чтобы заменить названное значение переменной joe.name значением "Mighty-Joe", можно выполнить следующий вызов:

```
joe.set_name("Mighty-Joe");
```

Однако обращаться с такими унаследованными переменными-членами, как name, нужно очень внимательно. Переменная-член name класса HourlyEmployee унаследована от класса Employee, но она является закрытым членом базового класса, то есть непосредственный доступ к ней возможен только из определений функций-членов класса Employee. Закрытая переменная-член (или функция-член) базового класса не доступна по имени в определении функции-члена любого класса, будь то посторонний класс или даже класс, производный от данного базового класса. И хотя класс HourlyEmployee содержит переменную-член name (унаследованную от базового класса Employee), к ней нельзя непосредственно обращаться в определениях функций-членов производного класса.

В качестве примера ниже приведены несколько начальных строк тела функции-члена HourlyEmployee::print_check из листинга 16.5:

```
void HourlyEmployee::print_check()
{
    set_net_pay(hours * wage_rate);

    cout << "\n_____\n";
    cout << "Pay to the order of " << get_name() << endl;
    cout << "The sum of " << get_net_pay() << " Dollars\n";
```

Наверняка у вас возникнет вопрос: зачем значение переменной-члена net_pay устанавливать с помощью функции-члена set_net_pay, нельзя ли переписать этот фрагмент кода так:

```
void HourlyEmployee::print_check()
{ net_pay = hours * wage_rate;   // Недопустимое использование переменной net_pay.
```

Как показывает комментарий, это запрещено. Переменная-член net_pay является закрытой переменной класса Employee, и хотя производный класс HourlyEmployee ее наследует, обращаться к ней непосредственно он не может. Поэтому для доступа к этой переменной приходится пользоваться какой-нибудь открытой функцией-членом класса Employee. Правильное определение функции print_check класса HourlyEmployee приведено в листинге 16.5 (и его фрагмент был показан выше.)

То, что переменные name и net_pay являются закрытыми членами класса Employee, объясняет, почему в определении функции HourlyEmployee::print_check вместо

непосредственного использования имен name и net_pay вызываются функции-аксессоры get_name и get_net_pay. Закрытые переменные-члены класса Employee здесь нельзя упоминать по имени. Вместо этого следует использовать открытые функции-члены — аксессоры и мутаторы (такие, как get_name и set_name), определенные в базовом классе. (Напомним, что аксессоры и мутаторы — это функции, позволяющие соответственно прочитать и изменить значение переменной-члена класса. Они подробно рассмотрены в главе 6.)

Запрет доступа к закрытым переменным-членам базового класса в определениях функций-членов производного класса многим кажется неправильным. В конце концов, если вы сотрудник с почасовой оплатой и хотите изменить свое имя, никто не скажет вам: «Просим прощения, переменная name (то есть ваше имя) является закрытой переменной класса Employee». Ведь будучи сотрудником с почасовой оплатой, вы являетесь и просто сотрудником. Однако на самом деле правила использования закрытых переменных-членов и закрытых функций-членов должны быть именно такими, как описано в этом разделе, в противном случае их «закрытость» окажется под вопросом. Если закрытая переменная-член класса будет доступна в определениях функций-членов производных классов, то в любой момент, когда какой-нибудь программист захочет обратиться к закрытой переменной-члену, ему достаточно будет просто создать производный класс и делать там с закрытыми переменными-членами базового класса, что заблагорассудится. Использование такой возможности приводит, прежде всего, к случайным ошибкам из-за того, что программист ненамеренно может изменить значение закрытой переменной-члена недопустимым образом и тем самым нарушить корректную работу или правильную структуру объекта. (Вспомните, что функции доступа и изменения, в первую очередь, защищают от случайных недопустимых модификаций переменных-членов.)

Мы поговорим о возможных способах обхода ограничения на доступ к закрытым переменным-членам базового класса в подразделе «Модификатор protected».

Ловушка: закрытые функции-члены можно считать ненаследуемыми

Как говорилось ранее, непосредственный доступ к закрытой переменной-члену (или функции-члену) базового класса невыполним вне интерфейса и реализации базового класса, более того, он невозможен даже в определении функции-члена производного класса. Закрытые функции-члены недоступны, как и переменные, но в отношении первых это ограничение является даже более жестким. Ведь доступ к закрытой переменной-члену базового класса, в конце концов, возможен с помощью аксессора или мутатора, тогда как закрытая функция-член абсолютно недоступна, словно она и вовсе не унаследована.

Однако никакой проблемы в этом нет. Закрытые функции-члены следует использовать только как вспомогательные функции, именно поэтому их применение необходимо ограничить их собственным классом. Если же некоторая функция требуется в производном классе, это означает, что она не просто вспомогательная, а следовательно, должна быть открытой.

Модификатор protected

Как вы уже видели, закрытые переменные-члены и закрытые функции-члены базового класса недоступны в определении производного класса. Однако существует особая разновидность переменных-членов и функций-членов, доступных по именам в производном классе, но нигде больше. Эта разновидность членов класса задается с помощью модификатора protected, а не private или public, как закрытые и открытые члены класса. Если перед именем члена класса поместить ключевое слово protected, для любого класса или функции вне данного класса он будет таким же, как если бы был объявлен с модификатором private, но в производном классе будет доступен по имени. Такой член класса называется *защищенным*.

Рассмотрим в качестве примера класс HourlyEmployee, производный от класса Employee. Для работы с унаследованными переменными-членами в определении функции HourlyEmployee::print_check нам приходилось пользоваться аксессорами и мутаторами. Однако если все закрытые переменные-члены класса Employee объявить с ключевым словом protected, а не private, определение этой функции можно переписать так:

```
void HourlyEmployee::print_check()
// Будет работать только в том случае, если переменные-члены класса
// Employee объявлены с модификатором protected, а не private.
{
    net_pay = hours * wage_rate;

    cout << "\n_____\n";
    cout << "Pay to the order of " << name << endl;
    cout << "The sum of " << net_pay << " Dollars\n";
    cout << "_____\n";
    cout << "Check Stub: NOT NEGOTIABLE\n";
    cout << "Employee Number: " << ssn << endl;
    cout << "Hourly Employee. \nHours worked: " << hours
         << " Rate: " << wage_rate << " Pay: " << net_pay
         << endl;
     cout << "_____\n";
}
```

Если переменные-члены name, net_pay и ssn объявить в базовом классе Employee как защищенные, то в производном классе HourlyEmployee к ним можно будет обращаться по именам. Однако в любом другом классе, не являющемся производным от Employee, эти переменные будут восприниматься как закрытые.

Переменные-члены, объявленные в базовом классе с модификатором protected, в производных классах тоже являются защищенными. Предположим, мы определили производный от класса HourlyEmployee класс PartTimeHourlyEmployee, в этом случае он наследует все переменные-члены базового класса, включая и переменные-члены класса Employee. В результате класс PartTimeHourlyEmployee будет содержать переменные-члены name, net_pay и ssn. И если в классе Employee они объявлены с ключевым словом protected, то в классе PartTimeHourlyEmployee к ним можно будет обращаться по именам.

Во всех остальных местах программы, за исключением производных классов, защищенные переменные-члены класса воспринимаются как закрытые.

Мы рассказали о защищенных членах классов прежде всего потому, что вы будете сталкиваться с ними в программах других разработчиков. Но многие специалисты в области программирования считают использование защищенных членов класса плохим стилем. Они утверждают, что это нарушает принцип сокрытия реализации класса, и настаивают на том, что все переменные-члены класса следует объявлять с модификатором private. Если объявить все переменные-члены класса как закрытые, в определениях функций-членов производных классов они будут недоступны, однако это вовсе не так плохо, как кажется. Такие переменные доступны косвенно с помощью унаследованных функций, считывающих или изменяющих их значения. В завершение скажем, что поскольку даже авторитетные специалисты расходятся во мнении относительно того, следует ли пользоваться защищенными членами класса, вы можете поступать согласно собственным предпочтениям.

Защищенные члены

Если перед именем переменной-члена класса поместить модификатор protected, а не private или public, любые классы и функции вне данного класса и его потомков будут воспринимать ее как закрытую. Однако в определениях функций-членов производных классов к этой переменной можно будет обращаться по имени. То же самое касается и функций-членов класса: если перед именем функции-члена класса поместить модификатор protected, а не private или public, любые классы и функции вне данного класса и его потомков будут воспринимать ее как закрытую. Однако в определениях функций-членов производных классов можно будет пользоваться этой функцией. Члены класса, объявленные с ключевым словом protected, называется защищенными.

Члены, объявленные в базовом классе с модификатором protected, в производных классах тоже считаются защищенными. Иными словами, если член базового класса является защищенным, к нему можно обращаться по имени в определениях всех классов-потомков, а не только в классах, производных от него непосредственно.

Упражнения для самопроверки

1. Допустима ли следующая программа (если в ней имеются соответствующие директивы #include и using):

```
void show_employee_data(const Employee object);

int main()
{
    HourlyEmployee joe("Mighty Joe",
                       "123-45-6789", 20.50, 40);
    SalariedEmployee boss("Mr. Big Shot",
                       "987-65-4321", 10500.50);
    show_employee_data(joe);
    show_employee_data(boss);

return 0;
}

void show_employee_data(const Employee object)
{
    cout << "Name: " << object.get_name() << endl;
```

```
    cout << "Social Security Number: "
         << object.get_ssn() << endl;
}
```

2. Существует базовый класс Smart с таким определением:

```
class Smart
{
public:
    Smart();
    void print_answer() const;
protected:
    int a;
    int b;
};
```

Приведите определение производного от него класса SmartBut (директивы #include и пространства имен в данном случае вас не интересуют).

Этот класс должен содержать дополнительную переменную-член crazy типа bool, а также дополнительную функцию-член is_crazy без аргументов, возвращающую значение типа bool, и подходящие конструкторы. Реализацию данного класса приводить не нужно.

3. Допустимо ли в производном классе SmartBut из упражнения 2 следующее определение функции-члена is_crazy:

```
bool SmartBut::is_crazy() const
{
    if (a > b)
        return crazy;
    else
        return true;
}
```

Поясните свой ответ. (Учтите, вопрос заключается в допустимости определения, а не в его осмысленности.)

Переопределение функций-членов

В определении производного класса HourlyEmployee (см. листинг 16.3) объявлялись новые функции-члены set_rate, get_rate, set_hours и get_hours. Кроме того, в нем приводилось объявление одной из функций, унаследованных от класса Employee. Унаследованные функции-члены, не объявленные в производном классе (например, set_name и set_ssn), включаются в него без изменений. В классе HourlyEmployee они имеют те же определения, что и в классе Employee. В определении производного класса, подобного HourlyEmployee, приводятся объявления только тех унаследованных функций, определения которых в производном классе вы хотите изменить или написать заново. Так, в файле реализации класса HourlyEmployee, приведенном в листинге 16.5, содержится новое определение унаследованной функции-члена print_check. Эта функция переопределена и в классе SalariedEmployee, как показано в листинге 16.6. Более того, в классах HourlyEmployee и SalariedEmployee содержатся разные определения функции print_check, то есть эта функция переопределена в производных классах.

В листинге 16.7 приведена программа, демонстрирующая использование классов HourlyEmployee и SalariedEmployee.

Листинг 16.6. Реализация производного класса SalariedEmployee

```
// Это файл реализации: salariedemployee.cpp.
// Он содержит реализацию класса SalariedEmployee.
// Интерфейс класса SalariedEmployee находится
// в заголовочном файле salariedemployee.h.
#include <iostream>
#include <string>
#include "salariedemployee.h"
using namespace std;

namespace employeessavitch
{
    SalariedEmployee::SalariedEmployee() : Employee(), salary(0)
    {
        // Тело намеренно оставлено пустым.
    }

    SalariedEmployee::SalariedEmployee(string the_name, string the_number,
                                       double the_weekly_salary)
                    : Employee(the_name, the_number), salary(the_weekly_salary)
    {
        // Тело намеренно оставлено пустым.
    }

    double SalariedEmployee::get_salary() const
    {
        return salary;
    }

    void SalariedEmployee::set_salary(double new_salary)
    {
        salary = new_salary;
    }

    void SalariedEmployee::print_check()
    {
        set_net_pay(salary);
        cout << "\n_____\n";
        cout << "Pay to the order of " << get_name() << endl;
        cout << "The sum of " << get_net_pay() << " Dollars\n";
        cout << "_____\n";
        cout << "Check Stub NOT NEGOTIABLE \n";
        cout << "Employee Number: " << get_ssn() << endl;
        cout << "Salaried Employee. Regular Pay: "
             << salary << endl;
        cout << "_____\n";
    }
}   // employeessavitch
```

Листинг 16.7. Использование производных классов

```
#include <iostream>
#include "hourlyemployee.h"
#include "salariedemployee.h"
```

```
using std::cout;
using std::endl;
using namespace employeessavitch;

int main()
{
    HourlyEmployee joe;
    joe.set_name("Mighty Joe");
    joe.set_ssn("123-45-6789");
    joe.set_rate(20.50);
    joe.set_hours(40);
    cout << "Check for " << joe.get_name()
         << " for " << joe.get_hours() << " hours.\n";
    joe.print_check();
    cout << endl;

    SalariedEmployee boss("Mr. Big Shot", "987-65-4321", 10500.50);
    cout << "Check for " << boss.get_name() << endl;
    boss.print_check();

    return 0;
}
```

Пример диалога

```
Check for Mighty Joe for 40 hours.

_____________________________________________________
Pay to the order of Mighty Joe
The sum of 820 Dollars

_____________________________________________________
Check Stub: NOT NEGOTIABLE
Employee Number: 123-45-6789
Hourly Employee.
Hours worked: 40 Rate: 20.5 Pay: 820

_____________________________________________________
Check for Mr. Big Shot

_____________________________________________________
Pay to the order of Mr. Big Shot
The sum of 10500.5 Dollars

_____________________________________________________
Check Stub NOT NEGOTIABLE
Employee Number: 987-65-4321
Salaried Employee. Regular Pay: 10500.5
```

Переопределение и унаследованные функции

Производный класс наследует все функции-члены (и переменные-члены) базового класса. Однако если в производном классе требуется иная реализация унаследованной функции, ее можно переопределить. Когда функция-член переопределяется в производном классе, ее объявление нужно включить в определение этого класса, причем даже тогда, когда оно остается неизменным по сравнению с базовым классом. Если вы не хотите переопределять унаследованную функцию-член, то в определении производного класса ее упоминать не нужно.

Переопределение и перегрузка

Не следует путать *переопределение* функции в производном классе с ее *перегрузкой*. В случае переопределения функции в дочерний класс включается ее новое определение с тем же количеством и теми же типами параметров. Если же определенная в производном классе функция будет иметь другое количество параметров или же они будут принадлежать к другим типам, то в данном классе окажутся обе функции, и это будет называться перегрузкой. Предположим, что мы добавили в класс HourlyEmployee такое объявление функции:

```
void set_name(string first_name, string last_name);
```

Тогда этот класс содержит функцию set_name с двумя аргументами и унаследованную функцию set_name с одним аргументом:

```
void set_name(string new_name);
```

Таким образом, в нем оказываются две функции с одним именем — то есть функция set_name в нем перегружена.

С другой стороны, в обоих классах, Employee и HourlyEmployee, определена следующая функция:

```
void print_check();
```

Теперь в классе HourlyEmployee имеется единственная функция print_check, и ее определение отличается от определения в классе Employee. Это означает, что функция print_check в нем переопределена.

И перегрузка, и переопределение функции здесь допустимы, но важно, чтобы вы понимали суть этих процессов, различали их и умели эффективно использовать.

Доступ к переопределенной функции базового класса

Допустим, мы переопределили некоторую функцию, и теперь в производном классе она определена иначе, чем в базовом. В таком случае определение данной функции для объектов производного класса не потеряно, но чтобы его использовать, нужно указать, что нам требуется определение функции из базового класса. Для этого следует применить оператор доступа к члену класса по его имени. Рассмотрим пример.

Вновь обратимся к базовому классу Employee (см. листинг 16.1) и производному классу HourlyEmployee (см. листинг 16.3) — функция print_check определена в обоих классах. Теперь предположим, что у нас имеется по объекту каждого класса:

```
Employee jane_e;
HourlyEmployee sally_h;
```

Тогда в вызове

```
jane_e.print_check();
```

используется определение функции print_check из класса Employee, а в вызове

```
sally_h.print_check();
```

определение функции print_check из класса HourlyEmployee.

А теперь нам понадобилось вызвать версию функции print_check из класса Employee для объекта производного класса sally_h. Это можно сделать так:

```
sally_h.Employee::print_check();
```

Конечно, маловероятно, что нам потребуется данная функция, но для других классов и функций определение из базового класса может в некоторых случаях быть полезным. Один такой пример приведен в упражнении 6.

Сигнатура

Сигнатура функции — это ее имя с указанием типов параметров, но без ключевого слова const и без символа амперсанда (&). При перегрузке функции два ее определения должны иметь разные сигнатуры.

Если функция с одним и тем же именем имеет в базовом и производном классах разные сигнатуры, она перегружена, а не переопределена.

Упражнения для самопроверки

4. Класс SalariedEmployee наследует от базового класса Employee, помимо прочих, функции get_name и print_check, но в его определении приведено только объявление функции print_check. Почему в определении отсутствует объявление функции get_name?
5. Приведите определение класса TitledEmployee, производного от базового класса SalariedEmployee (см. листинг 16.4). У этого производного класса имеется дополнительная переменная-член title типа string и дополнительная функция-член get_title без аргументов, возвращающая значение типа string, и функция-член set_title типа void, принимающая один аргумент типа string. Кроме того, в нем переопределена функция set_name. Необходимо привести только реализацию этого класса. Укажите все необходимые директивы #include и using namespace. Включите класс TitledEmployee в пространство имен employeessavitch.
6. Приведите определение конструктора класса TitledEmployee из упражнения 5, а также укажите новое определение функции-члена set_name. Эта функция должна добавлять к имени сотрудника его должность. Директивами #include и вопросами пространства имен заниматься не нужно.

16.2. Подробнее о наследовании

Дьявол ловит нас на мелочах.

Пословица

В этом разделе описываются некоторые нюансы наследования. Большинство рассматриваемых здесь вопросов относится только к классам, в которых используются динамические массивы или указатели на другие динамические данные.

Функции, которые не наследуются

Если Derived — производный класс от базового класса Base, тогда все «нормальные» функции класса Base наследуются классом Derived. Однако существуют некоторые особые функции, которые не наследуются дочерними классами. Вы уже знаете о том, что таковыми являются конструкторы и закрытые функции-члены класса. Кроме того, не наследуются деструкторы.

Конструктор копирования тоже не наследуется, но если он не определен в производном классе (или любом другом классе), C++ сгенерирует его автоматически. Однако помните, что сгенерированный автоматически конструктор копирует только значения переменных-членов и не может правильно работать с классами, содержащими указатели. Поэтому если в переменных-членах класса хранятся указатели, динамические массивы или другие динамические данные, следует обязательно определить для этого класса конструктор копирования (причем независимо от того, является ли класс производным или просто самостоятельным).

Перегруженный оператор присваивания (=) тоже не наследуется. Если в базовом классе Base определен оператор присваивания, а в производном классе Derived он не определен, то класс Derived будет иметь оператор присваивания, но такой, который создан языком C++ по умолчанию и не имеет ничего общего с оператором = класса Base.

То, что конструкторы, деструкторы и операторы присваивания не наследуются, достаточно естественно, поскольку для правильного выполнения задач им необходима информация, которой базовый класс не обладает. В частности, им нужно «знать» о новых переменных-членах, появившихся в производном классе.

Операторы присваивания и конструкторы копирования в производных классах

Перегруженный оператор присваивания и конструкторы не наследуются. Однако они могут использоваться (и в большинстве случаев используются) в определениях перегруженных операторов присваивания и конструкторов копирования производных классов.

При перегрузке оператора присваивания производного класса обычно применяется перегруженный оператор присваивания базового класса. В этом разделе подробно рассказывается о том, как строится его код. Напомним, что перегруженный оператор присваивания должен быть определен как функция-член класса.

Если Derived — производный класс от базового класса Base, то определение перегруженного оператора присваивания для него обычно начинается со следующего кода:

```
Derived& Derived::operator =(const Derived& right_side)
{
    Base::operator =(right_side);
```

Первой строкой кода в теле определения функции является вызов перегруженного оператора присваивания класса Base. Этот оператор «позаботится» об унаследованных переменных-членах и их значениях. Далее в определении оператора

присваивания должны задаваться значения новых переменных-членов класса, добавленных в класс Derived.

Подобная ситуация имеет место и для определения конструктора копирования производного класса. Если класс Derived – производный от Base, то в определении конструктора копирования этого класса обычно используется конструктор копирования класса Base, устанавливающий значения переменных-членов. Определение конструктора копирования производного класса чаще всего начинается с такого кода:

```
Derived::Derived(const Derived& object)
                : Base(object), ... Возможна дополнительная инициализация. ...
{
```

Вызов конструктора копирования базового класса Base(object) служит для установки в созданном объекте переменных-членов, унаследованных от базового класса. Обратите внимание, что объект object имеет тип Derived, поэтому он относится также к типу Base и, таким образом, является допустимым аргументом конструктора копирования класса Base.

Конечно, эти технологии не будут работать без правильно функционирующих оператора присваивания и конструктора копирования базового класса. То есть необходимо, чтобы в определении базового класса содержался конструктор копирования. И тогда либо автоматически созданный для этого класса оператор присваивания должен работать правильно, либо класс должен содержать перегруженное определение оператора присваивания.

Деструкторы и производные классы

Если базовый класс содержит правильно функционирующий деструктор, то определить деструктор производного класса обычно относительно легко. При вызове деструктора производного класса деструктор базового класса вызывается автоматически, и поэтому явный оператор вызова деструктора базового класса не требуется. А раз так, деструктору производного класса нужно только «позаботиться» об использовании оператора delete для неунаследованных переменных-членов, поскольку для унаследованных переменных-членов этот оператор будет вызван деструктором базового класса.

Если класс B является производным от класса A и класс C является производным от класса B, то при выходе объекта класса C из области видимости программы сначала вызывается деструктор класса C, затем деструктор класса B и потом деструктор класса A. Здесь обратите внимание, что деструкторы вызываются в порядке, обратном вызову конструкторов.

Упражнения для самопроверки

7. Вам известно, что перегруженный оператор присваивания и конструктор копирования не наследуются. Означает ли это, что в случае, когда не определен перегруженный оператор присваивания или конструктор копирования для производного класса, он не будет иметь этого члена?

8. Предположим, что класс Child — производный от класса Parent, а класс Grandchild — от класса Child. Следующие вопросы связаны с конструкторами и деструкторами всех трех классов. Какие конструкторы и в каком порядке вызываются при вызове конструктора класса Grandchild? Какие деструкторы и в каком порядке вызываются при вызове деструктора класса Grandchild?

9. Приведите определения функции-члена add_value, конструктора копирования, перегруженного оператора присваивания и деструктора для следующего класса. Этот класс предназначен для работы с частично заполненным массивом. В переменной-члене number_used содержится количество заполненных элементов массива. Для облегчения задания приведено определение другого конструктора.

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

class PartFilledArray
{
public:
    PartFilledArray(int array_size);
    PartFilledArray(const PartFilledArray& object);
    ~PartFilledArray();
    void operator = (const PartFilledArray& right_side);
    void add_value(double new_entry);
    // У этого класса может быть больше функций-членов,
    // но к данному упражнению они отношения не имеют.
protected:
    double *a;
    int max_number;
    int number_used;
};

PartFilledArray::PartFilledArray(int array_size)
                    : max_number(array_size), number_used(0)
{
    a = new double[max_number];
}
```

(Многие специалисты считают, что переменные-члены класса должны быть закрытыми, а не защищенными. Мы склонны с ними согласиться, однако попрактиковаться в использовании защищенных членов класса все же полезно, хотя бы потому, что ими пользуются некоторые другие программисты.)

10. Определите производный от класса PartFilledArray класс PartFilledArrayWMax. У него должна быть одна дополнительная переменная-член max_value для максимального из хранящихся в массиве значений. Определите аксессор get_max, возвращающий максимальное из хранящихся в массиве значений. Переопределите функцию-член add_value и определите два конструктора, у одного из которых будет аргумент типа int для максимального количества элементов массива. Определите также конструктор копирования, перегруженный оператор присваивания и деструктор. (Настоящий класс должен содержать больше функций-членов, но в этом упражнении они не нужны.)

16.3. Полиморфизм

Я делал это по-своему.
Фрэнк Синатра

В терминах объектно-ориентированного программирования полиморфизм представляет собой возможность связывать несколько задач с одним именем функции. Однако в настоящее время он принял более узкое значение. Полиморфизм — это свойство, которое позволяет использовать одно и то же имя функции для решения двух или более схожих, но технически разных, задач при помощи специального механизма, называемого поздним связыванием. Полиморфизм является одним из ключевых компонентов философии ООП. Данным двум темам, позднему связыванию и полиморфизму, и посвящен следующий раздел.

Позднее связывание

Виртуальной функцией называется функция, которая, еще не будучи определенной каким-либо образом, может использоваться. Например, в графической программе мы применяем несколько видов геометрических фигур, таких как прямоугольники, окружности, овалы и т. п., и при этом каждая фигура является объектом своего класса. Так, класс `Rectangle` представляет прямоугольник и содержит переменные-члены, значения которых являются высотой, шириной и координатами центральной точки, а класс `Circle` представляет окружность и содержит переменные-члены, значения которых являются радиусом и координатами центральной точки. В правильно разработанном проекте все они, скорее всего, будут потомками одного родительского класса, называемого, скажем, `Figure`. Теперь предположим, что вы хотите написать функцию, рисующую на экране фигуру. Для создания окружности в ней понадобятся операторы, отличные от операторов для черчения прямоугольника, поэтому каждый класс будет иметь свою функцию рисования. Однако поскольку эти функции принадлежат классам, все они могут называться `draw`. Если `r` — объект типа `Rectangle`, а `c` — объект типа `Circle`, то `r.draw()` и `c.draw()` могут быть функциями, реализованными с помощью разного кода. Все это вам известно, но теперь мы поговорим о том, что действительно для вас ново, — о виртуальных функциях, определенных для родительского класса `Figure`.

Родительский класс может содержать функции, применимые ко всем фигурам. Например, в нем может быть функция `center`, перемещающая фигуру в центр экрана, при этом стирая ее и рисуя снова по центру. Для повторного вывода фигуры в центре экрана функция `Figure::center` может использовать функцию `draw`.

Но если вы задумаетесь о том, как использовать унаследованную функцию `center` с объектами классов `Rectangle` и `Circle`, то увидите одну сложность. Чтобы лучше понять, в чем она заключается, предположим, что класс `Figure` уже написан и даже некоторое время используется и что мы добавили класс для совершенно новой фигуры, скажем, треугольника, назвав его `Triangle`. Разумеется, этот класс будет производным от класса `Figure` и поэтому унаследует функцию `center`. Соответственно, данная функция должна применяться и правильно работать для

всех объектов класса Triangle. Вот тут и начинаются сложности. Функция center использует функцию draw, которая у каждого типа фигур своя. В унаследованной функции center будет использоваться определение функции draw, входящее в состав класса Figure. И для треугольников, то есть объектов класса Triangle, эта функция правильно работать не будет. Нам же нужно, чтобы унаследованная функция center вызывала функцию Triangle::draw, а не Figure::draw. Возможно ли это, если класс Triangle и функция Triangle::draw во время написания и компиляции функции Figure::center еще не существовали? Оказывается — возможно, если draw определена как *виртуальная функция*.

Определяя функцию как виртуальную, вы тем самым говорите компилятору: «Я не знаю, как эта функция реализована. Подожди, пока ее нужно будет использовать в программе, и тогда возьми реализацию из экземпляра объекта». Эта технология, когда определение реализации процедуры откладывается до времени выполнения программы, называется *поздним* или *динамическим связыванием*. Виртуальные функции являются средством позднего связывания, предоставляемого программистам языком C++. Ну а теперь рассмотрим все это на примере, в котором заодно показано, как пользоваться виртуальными функциями в программе. Для пояснения некоторых деталей использования виртуальных функций в языке C++ воспользуемся упрощенным примером из области, не связанной с рисованием фигур.

Виртуальные функции в языке C++

Предположим, мы разрабатываем программу для учета продаж в автомагазине. Эта программа должна быть гибкой, но мы не уверены, что способны предусмотреть все возможные ситуации, например все возможные типы продаж. Прежде всего, существуют обычные продажи розничным покупателям, которые приходят в магазин и покупают одну запчасть. Однако позже, вероятно, понадобится добавить продажи со скидками или ввести заказы по почте с доставкой. Все это будут продажи одной единицы товара. Для единичной продажи счет будет содержать только цену товара, но если добавить скидку, некоторые виды счетов будут зависеть от размера скидки. Программа должна вычислять сумму продаж за день, которая, естественно, равна сумме всех выписанных за день счетов. Возможно, нас заинтересует наибольшая, наименьшая и средняя суммы продаж за день. Все эти данные можно вычислять на основе отдельных счетов, но функции для вычисления суммы по каждому счету будут добавлены позже, когда мы определимся со всеми типами счетов, которые будут использоваться. Поэтому пока определим функцию для вычисления суммы счета как виртуальную. (Для простоты в этом первом примере будем считать, что каждая продажа включает только один элемент, хотя с помощью производных классов и виртуальных функций можно было бы добавить и продажи из нескольких элементов.)

В листингах 16.8 и 16.9 приведены интерфейс и реализация класса Sale, а все типы продаж будут производными от этого класса. Класс Sale соответствует простой продаже из одного элемента без скидок и налоговых отчислений. Обратите внимание на ключевое слово virtual в объявлении функции bill (листинг 16.8).

Эта функция используется в функции-члене savings и перегруженном операторе < (листинг 16.9). Поскольку она объявлена как виртуальная, позднее можно будет определять производные классы от класса Sale и в них задавать собственные версии функции bill, соответствующие объектам производных классов.

Листинг 16.8. Интерфейс базового класса Sale

```
// Это заголовочный файл sale.h.
// В нем содержится интерфейс класса Sale, который
// представляет собой класс для простейшей продажи.
#ifndef SALE_H
#define SALE_H

#include <iostream>
using namespace std;

namespace salesavitch
{

    class Sale
    {
    public:
        Sale();
        Sale(double the_price);
        virtual double bill() const;
        double savings(const Sale& other) const;
        // Возвращает значение, соответствующее сэкономленной сумме денег.
    protected:
        double price;
    };

    bool operator < (const Sale& first, const Sale& second);
    // Сравнивает две продажи, чтобы узнать, сумма какой из них больше.

}   // salesavitch

#endif   // SALE_H
```

Листинг 16.9. Реализация базового класса Sale

```
// Это файл реализации: sale.cpp
// Он содержит реализацию класса Sale.
// Интерфейс этого класса содержится
// в заголовочном файле sale.h.
#include "sale.h"

namespace salesavitch
{

    Sale::Sale() : price(0)
    {}

    Sale::Sale(double the_price) : price(the_price)
    {}

    double Sale::bill() const
    {
        return price;
```

продолжение ⇨

Листинг 16.9 (*продолжение*)

```
    }

    double Sale::savings(const Sale& other) const
    {
        return (bill() - other.bill());
    }

    bool operator < (const Sale& first, const Sale& second)
    {
        return (first.bill() < second.bill());
    }

} //salesavitch
```

Например, в листинге 16.10 приведен производный класс DiscountSale, в котором требуется особое определение функции bill. Когда функция-член savings и перегруженный оператор < используются с объектом DiscountSale, они вызывают ту версию функции bill, которая входит в состав класса DiscountSale. Это одна из замечательнейших возможностей, предоставляемых языком C++ для действительно гибкого и эффективного программирования. Рассмотрим вызов функции d1.savings(d2) для объектов d1 и d2 класса DiscountSale. Определение функции savings (в том числе и для объектов класса DiscountSale) приведено в файле реализации базового класса Sale, откомпилированного до создания класса DiscountSale. Однако когда в вызове функции d1.savings(d2) приходит время вызова функции bill, C++ уже имеет достаточно информации для того, чтобы использовать определение функции bill из класса DiscountSale.

Листинг 16.10. Интерфейс производного класса DiscountSale

```
// Файл discountsale.h
// Это интерфейс класса DiscountSale.
#ifndef DISCOUNTSALE_H
#define DISCOUNTSALE_H
#include "sale.h"

namespace salesavitch {
   class DiscountSale : public Sale
   {
   public:
      DiscountSale();
      DiscountSale(double the_price, double the_discount);
      // Скидка (the_discount) задана в виде процента от цены (the_price).
      virtual double bill() const; // Ключевое слово virtual здесь не обязательно, но хороший
                                   // стиль программирования предполагает его наличие.
    protected:
        double discount;
    };
}   // salesavitch
#endif   // DISCOUNTSALE_H


// Файл discountsale.cpp.
// Это реализация класса DiscountSale.
#include "discountsale.h"
```

```
namespace salesavitch {
    DiscountSale::DiscountSale() : Sale(), discount(0)
    {}

    DiscountSale::DiscountSale(double the_price, double the_discount)
            : Sale(the_price), discount(the_discount)
    {}

    double DiscountSale::bill() const
    {
        double fraction = discount/100;
        return (1 - fraction)*price;
    }
}   // salesavitch
```

Как же все это работает? Для того чтобы писать программы на C++, можно, конечно, считать, что все происходит само собой. Однако во введении к данному разделу были сделаны необходимые пояснения. Объявляя функцию как `virtual`, программист говорит компилятору C++: «Подожди, пока эту функцию нужно будет использовать в программе, а затем возьми реализацию из экземпляра объекта».

В листинге 16.11 приведена программа, демонстрирующая работу виртуальной функции `bill` и использующих ее функций.

Листинг 16.11. Использование виртуальной функции

```
// Демонстрирует работу виртуальной функции bill.
#include <iostream>
#include "sale.h"    // В этой директиве нет необходимости, но благодаря использованию
                     // директивы ifndef мы можем ее включить.
#include "discountsale.h"
using namespace std;
using namespace salesavitch;

int main()
{
    Sale simple(10.00);   // Один элемент за $10.00.
    DiscountSale discount(11.00, 10);   // Один элемент за $11.00 со скидкой 10%.

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);

    if (discount < simple)
    {
        cout << "Discounted item is cheaper.\n";
        cout << "Savings is $" << simple.savings(discount) << endl;
    }
    else
        cout << "Discounted item is not cheaper.\n";

    return 0;
}
```

Пример диалога

```
Discounted item is cheaper.
Savings is $0.10
```

Для использования виртуальных функций в C++ необходимо знать следующее:

- если в производном классе у функции будет иное определение, чем в базовом, и вам понадобится сделать ее виртуальной, то в объявление этой функции в базовом классе нужно добавить ключевое слово `virtual`. В производном классе в объявлении этой функции ключевое слово `virtual` не требуется. Если в базовом классе функция объявлена как виртуальная, то в производном классе она становится виртуальной автоматически (однако полезно все же пометить ее ключевым словом `virtual`, хотя это и не обязательно);
- зарезервированное слово `virtual` добавляется в объявление функции, но не в ее определение;
- без использования ключевого слова `virtual` нельзя создать виртуальную функцию и воспользоваться ее преимуществами.

Если виртуальные функции настолько хороши, то почему бы не объявлять таким образом все функции класса? В данном случае вопрос заключается в эффективности: для определения и вызова виртуальных функций компилятору и среде времени выполнения приходится делать гораздо больше работы, поэтому объявление всех функций как виртуальных снижает производительность программы.

Подмена функции

Когда определение виртуальной функции изменяется в производном классе, программисты часто говорят, что оно *подменено*. В книгах по языку C++ между терминами *переопределение* и *подмена* иногда делается различие. Оба термина относятся к изменению определения функции в производном классе. Если функция является виртуальной, говорят о ее подмене, если же функция не является виртуальной, говорят о ее переопределении. И хотя в обоих случаях программист делает одно и то же — пишет для функции новое определение в производном классе, — для компилятора это две совершенно разные ситуации.

Полиморфизм

Под полиморфизмом понимается возможность ассоциирования нескольких задач с одним именем функции средствами позднего связывания. Поэтому полиморфизм, позднее связывание и виртуальные функции — это собственно одна и та же тема.

Упражнение для самопроверки

11. Предположим, вы модифицировали определение класса `Sale` (приведенное в листинге 16.8), удалив зарезервированное слово `virtual`. Как изменится вывод программы в листинге 16.11?

Виртуальные функции и расширение совместимости типов

В этом разделе рассматриваются возможности, возникающие при объявлении функций-членов класса с ключевым словом virtual, и приводится пример их использования.

Язык C++ является строго типизированным — это означает, что в нем всегда проверяются типы элементов и выводится сообщение об ошибке в случае их несоответствия, такого как несоответствие типов между аргументом и формальным параметром, когда автоматическое приведение типов невозможно. Кроме того, это означает, что присвоенное переменной значение должно соответствовать типу этой переменой, хотя в некоторых строго определенных случаях C++ выполняет автоматическое преобразование типов, называемое приведением, когда присваиваемое переменной значение и сама переменная могут принадлежать к разным типам данных.

Например, C++ позволяет присвоить значение типа char или int переменной типа double. Однако этот язык не позволяет присвоить значение типа double или float переменной целочисленного типа (int, char, short, long).

Такой строгий контроль типов находится в противоречии с идеей объектно-ориентированного программирования. В данном разделе рассматривается подход к соответствию типов, принятый в C++ для классов и их объектов. Допустим, что у нас имеются классы A и B и в программе определены объекты этих классов. Присваивание между этими объектами возможно не всегда. В качестве примера предположим, что программа содержит следующее объявление типов:

```
class Pet   // Домашнее животное
{
public:
    virtual void print();
    string name;   // Имя
};

class Dog : public Pet   // Собака
{
public:
    virtual void print();   // Ключевое слово virtual не
                            // обязательно но оно вносит ясность.
                            // (Кроме того, это хороший стиль программирования!)
    string breed;   // Порода
};

Dog vdog;
Pet vpet;
```

Теперь сосредоточимся на членах name и breed. (Для того чтобы упростить этот пример, мы сделали их открытыми. Однако в реальном приложении они должны быть закрытыми, а для доступа к ним необходимы аксессоры и мутаторы.)

Всякая собака является домашним животным, и всякий объект типа Dog одновременно является объектом типа Pet. Поэтому можно предположить, что значения

типа Dog программа должна рассматривать как значения типа Pet и допускать следующие присваивания:

```
vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
```

Действительно, C++ допускает это. Значение типа класса, скажем, vdog, можно присвоить переменной типа родительского класса, например vpet, но обратное присваивание недопустимо. Кроме того, при выполнении указанного присваивания объект теряет член breed — у объекта vpet его не будет. Это называется *проблемой расщепления*. Попытка доступа к переменной-члену breed

```
cout << vpet.breed; // Недопустимо: у класса Pet нет переменной-члена breed.
```

приведет к ошибке. Разумеется, можно сказать, что это имеет смысл, поскольку после помещения в переменную vpet объект типа Dog становится объектом типа Pet и больше не имеет свойств, специфических для типа Dog. Тут можно спорить, но в любом случае при программировании приходится учитывать описанный нюанс и следить, чтобы он не вызвал ошибок в программе.

Иное дело, что в определенных случаях это может быть неудобно с точки зрения логики программы. К счастью, C++ позволяет интерпретировать объект типа Dog как объект типа Pet, не теряя его специфических свойств. Для этого используются указатели на динамические экземпляры объектов. Предположим, в программу добавлены следующие объявления:

```
Pet *ppet;
Dog *pdog;
```

Воспользовавшись указателями и динамическими переменными, можно интерпретировать щенка Tiny как объект типа Pet, не теряя его породы. C++ допускает следующий код:

```
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great Dane";
ppet = pdog;
```

Более того, через указатель ppet доступна переменная-член breed. Допустим, что функция Dog::print() определена следующим образом:

```
// Используем библиотеку классов iostream.
void Dog::print()
{
    cout << "name: " << name << endl;
    cout << "breed: " << breed << endl;
}
```

Тогда оператор

```
ppet->print();
```

выведет на экран следующее:

```
name: Tiny
breed: Great Dane
```

Это возможно благодаря тому, что print() является виртуальной функцией-членом. Программа, содержащая все перечисленные операторы, приведена в листинге 16.12.

Листинг 16.12. Наследование и виртуальные функции

```
// Программа, демонстрирующая использование виртуальной
// функции для решения проблемы расщепления объекта.

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

class Pet
{
public:
    virtual void print();
    string name;
};

class Dog : public Pet
{
public:
   virtual void print(); // Ключевое слово virtual не является необходимым, но оно делает
                         // код более понятным. (Кроме того, это хороший стиль программирования!)
string breed;
};

int main()
{
    Dog vdog;
    Pet vpet;

    vdog.name = "Tiny";
    vdog.breed = "Great Dane";
    vpet = vdog;

    // Оператор vpet.breed; недопустим, поскольку у класса Pet нет члена breed.

    Dog *pdog;
    pdog = new Dog;


    pdog->name = "Tiny";
    pdog->breed = "Great Dane";

    Pet *ppet;
    ppet = pdog;
    ppet->print();   // Эти два вызова выводят одно и то же:
    pdog->print();   // name: Tiny breed: Great Dane

    // Следующий код, обращающийся к переменным-членам
    // непосредственно,а не через виртуальные функции,
    // вызовет ошибку:
    // cout << "name: " << ppet->name << "  breed: "
    // << ppet->breed << endl;
```

продолжение ⇩

Листинг 16.12 (*продолжение*)

```
    // Будет сгенерировано следующее сообщение об ошибке:
    // 'class Pet' has no member named 'breed'.
    // См. раздел "Ловушка: без функций-членов не обойтись".

    return 0;
}

void Dog::print()
{
    cout << "name: " << name << endl;
    cout << "breed: " << breed << endl;
}

void Pet::print()
{
    cout << "name: " << endl;
    // Обратите внимание, что член breed не упоминается.
}
```

Пример диалога

```
name: Tiny
breed: Great Dane
name: Tiny
breed: Great Dane
```

Ловушка: проблема расщепления

Хотя объект производного класса можно присвоить переменной базового класса, при этом происходит так называемое расщепление объекта с потерей данных. Все члены-данные объекта производного класса, не определенные в базовом классе, в результате присваивания будут потеряны. Точно так же ни одна функция-член, не определенная в базовом классе, не будет доступна в результирующем объекте базового класса.

Если выполнить следующий код:

```
Dog vdog;
Pet vpet;
vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
```

переменную vpet нельзя будет использовать как вызывающий объект для функции-члена, введенной в классе Dog, и значение переменной-члена Dog::breed утратится.

Ловушка: без функций-членов не обойтись

Для того чтобы воспользоваться описанной выше возможностью расширения типа, необходимы виртуальные функции-члены. В качестве примера предположим, что вместо вызова

```
ppet->print();
```

использован следующий код:

```
cout << "name: " << ppet->name
     << " breed: " << ppet->breed << endl;
```

Казалось бы, он должен ввести то же, что и функция print, но на самом деле он вызовет сообщение об ошибке. Дело в том, что тип выражения

```
*ppet
```

определяется типом указателя ppet. Это указатель для класса Pet, а у данного класса отсутствует переменная-член breed.

Иное дело — функция print(). Она объявлена как виртуальная функция базового класса Pet. Поэтому, встретив вызов

```
ppet->print();
```

компилятор выясняет, что переменная ppet указывает на объект типа Dog, и выполняет вызов

```
Dog::print()
```

а не

```
Pet::print()
```

Объектно-ориентированное программирование с использованием динамических переменных — это область, требующая особого мышления. Поначалу в ней трудно ориентироваться и весьма полезно помнить два простых правила.

1. Если типом указателя p_ancestor является базовый класс типа указателя p_descendant, то допустимо следующее присваивание:

   ```
   p_ancestor = p_descendant;
   ```

   Более того, в результате этого присваивания не будет потерян ни один из членов-данных или членов-функций динамической переменной, на которую указывает p_descendant.

2. Хотя все дополнительные поля динамической переменной будут на месте, для доступа к ним необходима виртуальная функция-член.

Ловушка: попытка откомпилировать определение класса без определений всех его виртуальных функций-членов

Наилучшей стратегией любой разработки является постепенное наращивание. Это означает, что нужно написать небольшое количество кода, протестировать его, добавить еще немного кода, еще протестировать и т. д. Однако если попытаться откомпилировать класс с виртуальными функциями без реализации каждой из них, ничего не получится — компилятор выведет какие-то невразумительные сообщения об ошибках, причем даже в том случае, если эти функции-члены нигде не вызываются.

Когда хотя бы одна виртуальная функция-член не реализована, компилятор откажется компилировать класс и выдаст сообщение об ошибке вроде следующего:

«undefined reference to Class_Name virtual table» («не определена ссылка на виртуальную таблицу класса Class_Name»). И даже если у данного класса нет ни одного производного класса, а виртуальная функция-член только одна, все равно это сообщение будет выводиться, пока вы не напишете определение функции.

А малопонятным это сообщение становится из-за того, что без определений виртуальных функций компилятор может выдавать и другие сообщения об ошибках, «жалуясь» на неопределенные ссылки на используемые по умолчанию конструкторы, даже если эти конструкторы уже определены.

Совет программисту: объявляйте деструкторы как виртуальные

Существует хорошая практика всегда объявлять деструкторы виртуальными. Но прежде чем объяснить, почему это хорошо, мы расскажем кое-что о деструкторах и указателях, а также о том, что такое виртуальный деструктор.

Рассмотрим следующий код, в котором SomeClass — это класс, содержащий деструктор

```
SomeClass *p = new SomeClass;
    ...
delete p;
```

не являющийся виртуальным.

При вызове оператора delete для динамической переменной, на которую ссылается указатель p, автоматически вызывается деструктор класса SomeClass. Теперь посмотрим, что произойдет, если этот деструктор объявлен как виртуальный.

Механизм виртуальных функций действует в отношении деструкторов таким образом, словно деструкторы класса и всех его потомков имеют одинаковые имена (хотя на самом деле это не так). Предположим, что класс Derived является производным от класса Base и что деструктор класса Base объявлен как virtual. Теперь рассмотрим следующий код:

```
Base *pBase = new Derived;
    ...
delete pBase;
```

Когда оператор delete выполняется для указателя pBase, вызывается деструктор. Поскольку деструктор класса Base объявлен как virtual и объект, на который указывает pBase, имеет тип Derived, вызывается деструктор класса Derived (который, в свою очередь, вызывает деструктор класса Base). Если бы деструктор класса Base не был объявлен как virtual, был бы вызван только деструктор класса Base.

Важно также иметь в виду, что когда деструктор базового класса объявляется как virtual, все деструкторы производных классов автоматически объявляются так же (независимо от того, помечены ли они как virtual в своих классах). И в этом деструкторы ведут себя таким же образом, словно все они имеют одно и то же имя (хотя на самом деле это и не так).

Вот теперь можно объяснить, почему же деструкторы должны быть виртуальными. Предположим, у класса Base имеется переменная-член pB типа указателя и что конструктор класса Base создает динамическую переменную и присваивает ее адрес переменной pB, а деструктор класса Base удаляет эту динамическую переменную. При этом названный деструктор не объявлен как virtual. Кроме того, допустим, что класс Derived, производный от Base, имеет переменную-член pD типа указателя и что конструктор класса Derived создает динамическую переменную и присваивает ее адрес переменной pD, а деструктор класса Derived удаляет эту динамическую переменную. Рассмотрим такой код:

```
Base *pBase = new Derived;
    ...
delete pBase;
```

Поскольку деструктор базового класса не является виртуальным, вызывается только деструктор класса Base. Он возвращает в динамическую область память, занимаемую динамической переменной, на которую указывает pB, а память, занимаемая динамической переменной, на которую указывает pD, никогда не возвращается в динамическую область (до завершения работы программы.)

С другой стороны, если деструктор базового класса объявлен как виртуальный, при выполнении оператора delete для переменной pBase вызывается деструктор класса Derived (поскольку объект, на который указывает pBase, относится к классу Derived). Этот деструктор удаляет динамическую переменную, на которую указывает pD, а затем автоматически вызывает деструктор базового класса Base, удаляющий динамическую переменную, на которую указывает pB. Таким образом, когда деструктор базового класса объявлен как virtual, в динамическую область возвращается вся память, занимаемая динамическими переменными объекта. И поэтому на всякий случай лучше сразу объявлять все деструкторы как виртуальные.

Упражнения для самопроверки

12. Почему нельзя присвоить объект базового класса переменной производного класса?
13. Какая проблема связана с допустимым присваиванием объекта производного класса переменной базового класса?
14. Предположим, что и базовый и производный классы содержат функцию-член с одинаковой сигнатурой. В программе имеется указатель на объект базового класса, и с его помощью вызывается указанная выше функция-член. Какая из двух функций будет вызвана: член базового класса или член производного класса?

Резюме

- Наследование представляет собой технологию повторного использования кода путем создания одних классов на базе других с добавлением в них новых элементов.
- Объекты производного класса наследуют все члены базового класса и содержат дополнительные члены, определенные в их собственном классе.

- Позднее связывание означает, что решение о выборе версии функции-члена принимается во время выполнения программы. Для реализации позднего связывания в C++ используются виртуальные функции.
- Защищенный (объявленный как protected) член базового класса доступен в функциях-членах производного класса.

Ответы к упражнениям для самопроверки

1. Да. Объект производного класса можно использовать в качестве параметра типа базового класса. Объект типа HourlyEmployee является объектом типа Employee, и объект типа SalariedEmployee также является объектом типа Employee.

2. ```
   class SmartBut : public Smart
   {
   public:
       SmartBut();
       SmartBut(int new_a, int new_b, bool new_crazy);
       bool is_crazy() const;
   private:
       bool crazy;
   };
   ```

3. Допустимо, потому что переменные-члены a и b в базовом классе объявлены как защищенные, и следовательно, доступ к ним в производном классе возможен по именам. Если бы a и b были объявлены как закрытые, это было бы недопустимо.

4. Объявление функции get_name не приведено в определении класса SalariedEmployee, потому что эта функция не переопределяется в данном классе, она наследуется без изменений.

5. ```
   #include <iostream>
   #include "salariedemployee.h"
   using namespace std;
   namespace employeessavitch
   {
   class TitledEmployee : public SalariedEmployee
       {
       public:
           TitledEmployee();
           TitledEmployee(string the_name, string the_title,
                          string the_ssn, double the_salary);
           string get_tjtle() const;
           void set_title(string the_title);
           void set_name(string the_name);
       private:
           string title;
       };
   }  // employeessavitch
   ```

6.
```
namespace employeessavitch
{
    TitledEmployee::TitledEmployee()
            : SalariedEmployee(), title("No title yet")
    {
        // Тело намеренно оставлено пустым.
    }

    TitledEmployee::TitledEmployee(string the_name,
                        string the_title,
                        string the_ssn, double the_salary)
            : SalariedEmployee(the_name, the_ssn, the_salary),
                        title(the_title)
    {
        // Тело намеренно оставлено пустым.
    }

    void TitledEmployee::set_name(string the_name)
    {
        Employee::set_name(title + the_name);
    }
}   // employeessavitch
```

7. Нет, не означает. Если не определить перегруженный оператор присваивания или конструктор копирования, эти две функции будут автоматически определены для производного класса. Однако когда в этом классе используются указатели, динамические массивы или другие динамические данные, почти наверняка ни определенный по умолчанию конструктор копирования, ни определенный по умолчанию перегруженный оператор присваивания не будут удовлетворять потребностям класса.

8. Конструкторы вызываются в следующем порядке: сначала `Parent`, затем `Child` и наконец, `Grandchild`. Деструкторы вызываются в обратном порядке: сначала `Grandchild`, затем `Child` и потом `Parent`.

9.
```
// Используем библиотеки iostream и cstdlib.
void PartFilledArray::add_value(double new_entry)
{
    if (number_used == max_number)
    {
        cout << "Adding to a full array.\n";
        exit(1);
    }
    else
    {
        a[number_used] = new_entry;
        number_used++;
    }
}

PartFilledArray::PartFilledArray
          (const PartFilledArray& object)
                : max_number(object.max_number),
                    number_used(object.number_used)
{
    a = new double[max_number];
```

```
    for (int i = 0; i < number_used; i++)
        a[i] = object.a[i];
}

void PartFilledArray::operator =
                (const PartFilledArray& right_side)
{
    if (right_side.max_number > max_number)
    {
        delete [] a;
        max_number = right_side.max_number;
        a = new double[max_number];
    }
    number_used = right_side.number_used;

    for (int i = 0; i < number_used; i++)
        a[i] = right_side.a[i];
}

PartFilledArray::~PartFilledArray()
{
    delete [] a;
}
```

10.
```
class PartFilledArrayWMax : public PartFilledArray
{
public:
    PartFilledArrayWMax(int array_size);
    PartFilledArrayWMax(const PartFilledArrayWMax& object);
    ~PartFilledArrayWMax();
    void operator= (const PartFilledArrayWMax& right_side);
    void add_value(double new_entry);
    double get_max();
private:
    double max_value;
};

PartFilledArrayWMax::PartFilledArrayWMax(int array_size)
                        : PartFilledArray(array_size)
{
    // Тело функции намеренно оставлено пустым.
    // Переменная Max_value не инициализируется, поскольку
    // для нее нет подходящего значения по умолчанию.
}

/*
Обратите внимание, что следующий код не стал бы работать,
поскольку в нем вызывается используемый по умолчанию
конструктор класса PartFilledArray, но у этого класса
нет используемого по умолчанию конструктора:

PartFilledArrayWMax::PartFilledArrayWMax(int array_size)
                : max_number(array_size), number_used(0)
{
    a = new double[max_number];
}
*/
```

```
PartFilledArrayWMax::PartFilledArrayWMax
              (const PartFilledArrayWMax& object)
                : PartFilledArray(object)
{
    if (object.number_used > 0)
    {
        max_value = a[0];

        for (int i = 1; i < number_used; i++)
            if (a[i] > max_value)
                max_value = a[i];
    }   // В противном случае оставить переменную max_value неинициализированной.
}

// Приведенный код эквивалентен деструктору, создаваемому C++ по умолчанию,
// и поэтому данное определение можно опустить. В этом случае нужно будет
// удалить и объявление деструктора из определения класса.
PartFilledArrayWMax::~PartFilledArrayWMax()
{
    // Тело функции оставлено пустым.
}
void PartFilledArrayWMax::operator =
             (const PartFilledArrayWMax& right_side)
{
    PartFilledArray::operator = (right_side);
    max_value = right_side.max_value;
}

// Используем библиотеки iostream и cstdlib.
void PartFilledArrayWMax::add_value(double new_entry)
{
    if (number_used == max_number)
    {
        cout << "Adding to a full array.\n";
        exit(1);
    }

    if ((number_used == 0) || (new_entry > max_value))
        max_value = new_entry;

    a[number_used] = new_entry;
    number_used++;
}

double PartFilledArrayWMax::get_max()
{
    return max_value;
}
```

11. Вывод программы изменится так:

```
Discounted item is not cheaper.
```

12. Потому что у производного класса имеются дополнительные переменные-члены, которым нечего будет присваивать.

13. Хотя объект производного класса можно присвоить переменной базового класса, в этой переменной не окажется части объекта, определенной в производном классе. Это исчезновение части объекта называется проблемой расщепления.
14. Если функция базового класса объявлена как виртуальная, вызываемая функция определяется типом объекта, на который указывает указатель. Если же функция объявлена без ключевого слова `virtual`, вызываемая функция определяется типом переменной-указателя.

Практические задания

1. Напишите программу, в которой используется класс `SalariedEmployee`, определенный в листинге 16.4. Здесь должен определяться производный от него класс `Administrator`. Модификатор `private` в базовом классе можно заменить модификатором `protected`. Добавьте следующие переменные и функции:
   - переменную-член типа `string`, содержащую должность администратора (например, директор или вице-президент);
   - переменную-член типа `string`, содержащую область деятельности, за которую отвечает данный администратор (производство, учет, персонал);
   - переменную-член типа `string`, содержащую фамилию непосредственного начальника данного администратора;
   - защищенную переменную-член типа `double`, содержащую годовой оклад администратора. Если вы внесли предложенное выше изменение, можете использовать существующую переменную-член `salary`;
   - функцию-член `set_supervisor`, изменяющую фамилию непосредственного начальника;
   - функцию-член для ввода с клавиатуры данных об администраторе;
   - функцию-член `print`, выводящую данные объекта на экран;
   - перегруженную версию функции-члена `print_check()`, выводящую чек соответствующего содержания.
2. Добавьте в иерархию сотрудников, представленную определенными в листингах 16.1, 16.3 и 16.4 классами, классификацию сотрудников: временный, администратор, постоянный и т. п. Протестируйте все функции-члены. Для тестовой программы хорошо подойдет пользовательский интерфейс с меню.
3. Приведите определение класса `Doctor`, объекты которого представляют записи о врачах в клинике. Этот класс должен быть производным от класса `SalariedEmployee`, определенного в листинге 16.4. Запись `Doctor` содержит специальность врача: «педиатр», «акушер», «терапевт» и т. п. (тип `string`), а также стоимость консультации (тип `double`). Класс должен включать необходимый набор конструкторов, аксессоров и мутаторов, перегруженный оператор присваивания и конструктор копирования. Напишите отладочную программу для тестирования этого класса.

4. Создайте базовый класс Vehicle (транспортное средство), содержащий название производителя (тип string), количество цилиндров в моторе (тип int) и информацию о владельце (приведенный ниже тип Person). Затем создайте класс Truck (грузовик), производный от Vehicle, и добавьте в него следующие свойства: грузоподъемность в тоннах (тип double, поскольку это значение может содержать дробную часть) и массу в фунтах, которую грузовик может взять на буксир (тип int). Класс должен содержать необходимый набор конструкторов, аксессоров и мутаторов, перегруженный оператор присваивания и конструктор копирования. Напишите отладочную программу для тестирования этого класса.

```
class Person
{
public:
    Person();
    Person(string the_name);
    Person(const Person& the_object);
    string get_name() const;
    Person& operator=(const Person& rt_side);
    friend istream& operator >>(istream& in_stream,
                                Person& person_object);
    friend ostream& operator <<(ostream& out_stream,
                                const Person& person_object);
private:
    string name;
};
```

5. Приведите определение двух классов, Patient (пациент) и Billing (счет), объекты которых представляют записи в клинике. Класс Patient будет производным от класса Person, приведенного в задании 4. Объект Patient содержит имя пациента (унаследованное от класса Person) и объект класса Doctor (определенного в задании 3), представляющий личного терапевта. Объект типа Billing будет содержать объект Patient, объект Doctor и значение суммы оплаты типа double. Класс должен содержать необходимый набор конструкторов, аксессоров и мутаторов, перегруженный оператор присваивания и конструктор копирования. Сначала напишите отладочную программу для тестирования всех функций-членов, а затем отладочную программу, создающую две записи Billing и объекты, представляющие как минимум двух пациентов и двух врачей, и выводящую общий доход клиники на основе записей Billing.

6. Рассмотрим графическую систему с классами для различных фигур — прямоугольников, квадратов, треугольников, окружностей и т. п. Данные о прямоугольнике (высота, ширина и центральная точка), квадрате и окружности (только центральная точка, длина стороны или радиус) хранятся в соответствующих членах. В правильно спроектированной системе все классы фигур будут производными от класса Figure. Ваша задача — реализовать такую систему.

   Класс Figure является базовым. Из производных классов вам нужно создать только Rectangle и Triangle. Каждый класс содержит заглушки для функций-членов erase и draw. Эти две функции-члены выводят сообщение, указывающее, какая из функций вызвана и к какому классу относится ее вызывающий

объект. Поскольку это только заглушки, ничего больше они делать не должны. Функция-член center вызывает функции erase и draw для перемещения фигуры в центр экрана. Так как для этих функций имеются только заглушки, функция center не будет ничего перемещать, но она должна вызывать функции erase и draw. Добавьте в функцию center вывод сообщения о ее собственном вызове. У функций-членов не должно быть аргументов.

Это задание состоит из трех частей. Вы должны:

а) написать определения классов без использования виртуальных функций, откомпилировать и протестировать их;

б) сделать функции-члены базового класса виртуальными, откомпилировать и протестировать их;

в) объяснить разницу результатов.

В реальном примере вам пришлось бы заменить определение каждой функции кодом, выполняющим рисование. Для тестирования воспользуйтесь следующей функцией:

```
// Программа для тестирования классов, определенных в задании 6.
#include <iostream>
#include "figure.h"
#include "rectangle.h"
#include "triangle.h"
using std::cout;

int main()
{
    Triangle tri;
    tri.draw();
    cout <<
        "\nDerived class Triangle object calling center().\n";
    tri.center();   // Вызывает функции draw и center.

    Rectangle rect;
    rect.draw();
    cout <<
        "\nDerived class Rectangle object calling center().\n";
    rect.center();   // Вызывает функции draw и center.
    return 0;
}
```

7. Доработайте задание 6. Напишите определения для перечисленных конструкторов и функций-членов: Figure::center, Figure::draw, Figure::erase, Triangle::draw, Triangle::erase, Rectangle::draw и Rectangle::erase, чтобы функция draw действительно рисовала фигуры, помещая символ звездочки (*) в заданных местах экрана. Функции erase могут просто очищать экран (выводя пустые строки или каким-нибудь более сложным способом). В этой задаче множество деталей, и решение о том, какие из них применять, вам нужно будет принимать самостоятельно.

Глава 17 Обработка исключений

Исключение доказывает правило.
Известная максима

Можно писать программы, исходя из предположения, что ничего необычного или неправильного никогда не случится. Например, если программа считывает список значений, можно считать, что этот список никогда не бывает пустым. Создав программу, работающую в среде, где все и всегда происходит так, как планировалось, можно затем дополнить ее кодом, обрабатывающим нестандартные, исключительные ситуации. В языке C++ имеются средства для реализации такого подхода. Сначала пишется код для условий, в которых не происходит ничего необычного, а затем с помощью специальных средств обработки исключений добавляется код для нестандартных ситуаций.

Исключения широко используется для обработки ошибок, но гораздо правильнее рассматривать таковые как способ обработки «исключительных ситуаций». Ведь если программа правильно обработает «ошибку», то она перестанет быть ошибкой.

Пожалуй, самым важным применением исключений является разработка функций, ведущих себя по-разному в зависимости от того, для чего они вызваны. Так, функция может использоваться во множестве программ, одни из которых обрабатывают некоторое особое условие одним способом, а другие — иным. Скажем, если функция выполняет деление на нуль, в одной программе это должно вызвать немедленное завершение работы, а в другой — какие-нибудь иные действия. В этой главе вы увидите, что, встретив исключительное условие, такая функция может выбросить исключение, и тогда исключительную ситуацию можно будет обработать вне этой функции. Таким образом, для разных вызовов функции одна и та же исключительная ситуация обрабатывается по-разному.

В C++ обработка исключений выполняется так. Либо библиотечное программное обеспечение, либо ваш собственный код представляет механизм для подачи сигнала о том, что произошло нечто необычное. Такая подача сигнала называется *выбросом исключения* (иногда также инициированием, запуском или генерированием

исключения). В другое место программы помещается код, выполняющий действия, соответствующие исключительной ситуации. Эти действия именуются *обработкой исключения*. В результате созданный код программы будет эффективным, гибким и надежным. В этой главе подробно рассказывается о том, как все это можно сделать в программах на языке C++.

17.1. Основы обработки исключений

В большинстве ситуаций программа работает, и я не знал,
что она должна работать и в этом случае.

Аргументация студента, не согласного с оценкой

Обработка исключений занимает относительно малую часть программного кода приложений, и реальные ситуации, в которых она выполняется, слишком специфичны, чтобы рассказывать о них в одном примере. Поэтому мы продемонстрируем технологию обработки исключений в C++ на нескольких простых примерах, которые в реальной программе ее бы не требовали. Они удобны для обучения, но помните, что это лишь надуманные примеры и на практике в таких простых случаях обработка исключений не выполняется.

Простой пример обработки исключений

Предположим, что вы ежедневно обязательно потребляете молоко, и оно практически всегда имеется в вашем холодильнике. Программа должна учитывать такую маловероятную ситуацию, когда оно кончается. Базовый код, в котором предполагается, что молоко не может закончиться, мог бы быть таким:

```
cout << "Enter number of donuts:\n";
cin >> donuts;
cout << "Enter number of glasses of milk:\n";
cin >> milk;
dpg = donuts/static_cast<double>(milk);
cout << donuts << " donuts.\n"
     << milk << " glasses of milk.\n"
     << "You have " << dpg
     << " donuts for each glass of milk.\n";
```

Если молока не окажется, то здесь будет предпринята попытка деления на нуль, что является ошибкой. Для того чтобы предусмотреть особую ситуацию, когда молоко закончилось, можно добавить в программу условие проверки его количества. Полная программа с такой проверкой приведена в листинге 17.1. Она запрашивает у пользователя количество бубликов и стаканов молока и сообщает, сколько бубликов приходится на каждый стакан молока. Эта программа не выполняет обработку исключений. Далее мы посмотрим, как можно переписать ее с помощью средств обработки исключений, предоставляемых языком C++.

Листинг 17.1. Обработка особой ситуации без помощи исключений

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

int main()
{
    int donuts, milk;
    double dpg;
    cout << "Enter number of donuts:\n";
    cin >> donuts;
    cout << "Enter number of glasses of milk:\n";
    cin >> milk;

    if (milk <= 0)
    {
        cout << donuts << " donuts, and No Milk!\n"
             << "Go buy some milk.\n";
    }
    else
    {
        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
    }

    cout << "End of program.\n";
    return 0;
}
```

Пример диалога

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

В листинге 17.2 приведена новая версия программы, на этот раз с обработкой исключений. Подчеркнем еще раз, что это специально придуманный простой пример и для подобного кода в настоящей программе обработка исключений не требуется. Хотя программа не упростилась после модификации, ее часть между ключевыми словами `try` и `catch` стала четкой и понятной, и это показывает преимущества использования исключений. Давайте рассмотрим код этой части. В основном он такой же, как в листинге 17.1, но вместо громоздкого оператора `if...else` (он выделен) в новой программе содержится только следующий, намного меньший, оператор `if` (и несколько операторов без ветвления):

```
if (milk <=0)
    throw donuts;
```

Листинг 17.2. Обработка той же особой ситуации с использованием исключения

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

int main()
{
    int donuts, milk;
    double dpg;

    try
    {
        cout << "Enter number of donuts:\n";
        cin >> donuts;
        cout << "Enter number of glasses of milk:\n";
        cin >> milk;

        if (milk <= 0)
               throw donuts;

        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
    }
    catch(int e)
    {
        cout << e << " donuts, and No Milk!\n"
             << "Go buy some milk.\n";
    }

    cout << "End of program.\n";
    return 0;
}
```

Пример диалога 1

```
Enter number of donuts:
12
Enter number of glasses of milk:
6
12 donuts.
6 glasses of milk.
You have 2 donuts for each glass of milk.
End of program.
```

Пример диалога 2

```
Enter number of donuts:
12
Enter number of glasses of milk:
0
12 donuts, and No Milk!
Go buy some milk.
End of program.
```

Оператор `if` указывает, что при отсутствии молока следует сделать нечто исключительное. Эти исключительные действия определены после ключевого слова `catch`.

Основная идея обработки исключений в C++ такова: нормальная ситуация обрабатывается кодом, следующим за ключевым словом try, а код, располагающийся за ключевым словом catch, выполняется только в исключительных случаях. Таким образом нормальные ситуации отделяются от исключительных. В нашем примере это разделение не приносит особых преимуществ, но в других ситуациях оно может быть очень полезным. Давайте рассмотрим процесс подробнее.

Базовая технология обработки исключений в C++ заключается в использовании триады try...throw...catch. Блок try имеет следующий синтаксис:

```
try
{
    ... Некоторый код. ...
}
```

Здесь содержится код основного алгоритма, указывающий компьютеру, что делать, когда все идет по плану. Данный блок называется try (попробовать), потому что мы не полностью уверены, что все в порядке, но хотим попробовать действовать таким образом, «словно все в порядке». Если что-то пойдет не так, будет организован выброс исключения, то есть мы сообщим программе о проблемах. Это делается согласно следующей схеме:

```
try
{
    ... Выполняемый код. ...
    ... Возможный выброс исключения. ...
    ... Дополнительный код. ...
}
```

Вот пример блока try с оператором throw, скопированный из программы листинга 17.2:

```
try
    {
        cout << "Enter number of donuts:\n";
        cin >> donuts;
        cout << "Enter number of glasses of milk:\n";
        cin >> milk;

        if (milk <= 0)
               throw donuts;

        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
    }
```

Следующий оператор *выбрасывает* значение переменной donuts типа int:

```
throw donuts;
```

Выброшенное значение, в данном случае значение переменной donuts, иногда называют исключением, а выполнение оператора throw — *выбросом исключения*. Разрешается выбрасывать исключения любого типа. В этом случае используется значение типа int.

Оператор throw

Синтаксис

`throw` *выражение_для_выбрасываемого_значения*

Когда выполняется оператор `throw`, обработка блока `try`, в котором он находится, прекращается. Если за блоком `try` следует подходящий блок `catch`, управление передается ему. Оператор `throw` почти всегда выполняется в составе оператора ветвления, такого как `if`. Выбрасываемое значение может быть любого типа.

Пример

```
if (milk <= 0)
   throw donuts;
```

Когда нечто выбрасывается, оно перемещается из одного места в другое. В C++ перемещается поток управления (а также выброшенное значение). В программе на C++ при выбросе исключения выполнение блока `try` останавливается и начинается выполнение другой части кода, называемой блоком `catch`. Выполнение этого блока называется *перехватом* или *обработкой исключения*. Выброшенное исключение обязательно должно быть перехвачено и обработано некоторым блоком `catch`. В листинге 17.2 соответствующий блок `catch` следует непосредственно за блоком `try`. Вот он:

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

Данный блок очень похож на определение функции с параметром типа `int`. Однако это не определение функции, хотя в некотором роде блок `catch` можно было бы назвать функцией. Это отдельный фрагмент кода, выполняемый, когда программа достигает следующего оператора, расположенного внутри блока `try` (и выполняет его):

`throw` *переменная_int*;

Таким образом, этот оператор `throw` похож на вызов функции, но вместо функции он вызывает блок `catch`, после чего выполняется код этого блока. Блок `catch` часто называют *обработчиком исключений*, и этот термин тоже говорит о том, что его природа подобна функции.

Возникает вопрос, что же это за идентификатор `e` в следующей строке блока `catch`:

```
catch(int e)
```

Он похож на параметр, и в самом деле действует как параметр. Поэтому мы называем его *параметром блока* `catch`. (Но помните, что это еще не означает, что блок `catch` является функцией). Параметр блока `catch` выполняет следующие задачи.

1. Заданное перед параметром блока `catch` имя типа определяет значения, которые могут перехватываться этим блоком `catch`.
2. Этот параметр определяет имя выброшенного и перехваченного значения, по которому можно ссылаться на данное значение в коде блока.

Сейчас мы расскажем об использовании параметра блока catch в качестве имени выброшенного и перехваченного значения, а далее в подразделе «Выброс и перехват исключений различных типов» — о том, какой из имеющихся в программе блоков catch перехватит конкретное выброшенное значение. В рассматриваемом примере есть всего один блок catch. Традиционным именем параметра блока является e, но вместо него можно использовать любой допустимый идентификатор.

Теперь посмотрим, как работает блок catch в листинге 17.2. После выброса значения выполнение кода блока try прекращается, и управление передается блоку (или блокам) catch, следующему за блоком try. Вот какой блок catch используется в программе листинга 17.2:

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

Для того чтобы выброшенное программой значение могло быть обработано этим блоком catch, оно должно иметь тип int. В программе листинга 17.2 выбрасывается значение переменной donuts, а поскольку это переменная типа int, данный блок catch может перехватить значение.

Предположим, что названная переменная содержит значение 12, а переменная milk значение 0, как в примере из листинга 17.2. Так как значение второй переменной не положительное, выполняется оператор throw, вызываемый из оператора if... else. Этот оператор выбрасывает значение переменной donuts. Когда блок catch перехватывает значение, оно подставляется в параметр блока catch и выполняется код блока, выводящий на экран следующий текст:

```
12 donuts, and No Milk!
Go buy some milk.
```

Если значение переменной donuts положительно, оператор throw не выполняется, а просто выполняется до конца блок try, а по завершении его последнего оператора — код, расположенный после блока catch. Обратите внимание, что если исключение не выброшено, программа игнорирует блок catch.

Параметр блока catch

Параметр блока catch является идентификатором в заголовке блока, указывающим место подстановки значения (исключения), которое может быть выброшено программой. Когда (подходящее) значение выбрасывается в предшествующем блоке try, данное значение подставляется в параметр блока catch. Для этого параметра может использоваться любой допустимый идентификатор (не являющийся зарезервированным словом).

Пример

```
catch(int e)
{
    cout << e << " donuts, and No Milk!\n"
         << "Go buy some milk.\n";
}
```

В приведенном примере e является параметром блока catch.

Описанные действия конструкции try...throw...catch похожи на действия оператора if...else. Разница заключается только в том, что try...throw...catch дополнена возможностью *пересылки сообщения одной из ветвей*. Пока это различие не кажется таким уж серьезным, но далее вы увидите, насколько оно важно.

Сказанное можно подытожить так: блок try содержит некоторый код, в котором должен присутствовать оператор throw. Этот оператор выполняется только в исключительных ситуациях, его задача — выбросить значение определенного типа, называемое исключением. После выброса исключения (например, значения переменной donuts в программе листинга 17.2) выполнение блока try прекращается. Остальной код этого блока игнорируется, и управление передается соответствующему блоку catch. Блок catch применим только к предшествующему ему блоку try. После выброса исключения его значение подставляется в параметр блока catch, и выполняются операторы этого блока. Например, из диалогов листинга 17.2 видно, что как только пользователь вводит значение, не являющееся положительным, выполнение блока try прекращается и выполняется блок catch. Пока будем считать, что за каждым блоком try следует соответствующий блок catch, позднее мы расскажем, что происходит, если такого блока не оказывается.

Если в блоке try не выбрасывается исключение, то по завершении выполнения этого блока программа продолжает работу с оператора, следующего за блоком catch. Иными словами, если исключение не выбрасывается, блок catch игнорируется. В большинстве случаев оператор throw не обрабатывается и код блока try выполняется полностью, а код блока catch полностью игнорируется.

Конструкция try...throw...catch

Конструкция try...throw...catch — это базовый механизм выброса и перехвата исключений в языке C++. Оператор throw выбрасывает исключение (представляющее собой некоторое значение), а блок catch перехватывает его. После выброса исключения выполнение блока try прекращается и выполняется код блока catch. По завершении выполнения блока catch выполняется следующий за ним код (если этот блок не завершит выполнение программы или не выполнит другое специализированное действие).

Если в блоке try не будет выброшено исключение, после его завершения выполнение программы продолжится с оператора, следующего за блоком (блоками) catch. (Иначе говоря, если исключение не выброшено, блок (блоки) catch игнорируется.)

Синтаксис

```
try
{
    ... Некоторые операторы.
            Здесь поместите оператор throw или вызов функции,
            которая может выбросить исключение.
        Еще_операторы. ...
}
catch(имя_типа e)
{
    ... Код, выполняемый в случае перехвата данным блоком catch
            значения, выброшенного в блоке try. ...
}
```

Упражнения для самопроверки

1. Что выводит следующий код:

```
int wait_time = 46;
try
{
    cout << "Try block entered.\n";
    if (wait_time > 30)
        throw wait_time;
    cout << "Leaving try block.\n";
}

catch(int thrown_value)
{
    cout << "Exception thrown with\n"
         << "wait_time equal to " << thrown_value << endl;
}

cout << "After catch block." << endl;
```

2. Что выведет код из упражнения 1, если строку

```
int wait_time = 46;
```

заменить строкой

```
int wait_time = 12;
```

3. Какой оператор в упражнении 1 является оператором throw?
4. Что происходит при выполнении оператора throw? Дайте общий ответ на этот вопрос, а не просто опишите какой-либо пример кода.
5. Где в упражнении 1 блок try?
6. Найдите в упражнении 1 блок catch.
7. Где в упражнении 1 параметр блока catch?

Определение собственных классов исключений

Оператор throw может выбросить значение любого типа. Однако, как правило, в программе определяют класс, объекты которого могут передать в блок catch определенную информацию о причине исключения. Более того, для каждого типа исключительных ситуаций можно определить собственный, специализированный класс исключения.

Класс исключения — это просто класс. Исключением его делает способ применения, а не какая-то особая структура. Но выбор имени и структуры класса исключения должен осуществляться продуманно.

В листинге 17.3 приведен пример программы с определенным программистом классом исключения. Это опять-таки просто программа, иллюстрирующая некоторые детали обработки исключений в C++. В ней слишком много сложностей для той простой задачи, которую она выполняет, но в качестве примера она подходит нам как нельзя лучше, поскольку не отвлекает внимание на код, не относящийся к обработке исключений.

Листинг 17.3. Определение пользовательского класса исключений

```
// Это лишь пример для изучения синтаксиса C++. Не считайте его
// примером ситуации, в которой требуется обработка исключения.
#include <iostream>
using namespace std;
class NoMilk
{
public:
    NoMilk();
    NoMilk(int how_many);
    int get_donuts();
private:
    int count;
};

int main()
{
    int donuts, milk;
    double dpg;

    try
    {
        cout << "Enter number of donuts:\n";
        cin >> donuts;
        cout << "Enter number of glasses of milk:\n";
        cin >> milk;

        if (milk <= 0)
              throw NoMilk(donuts);

        dpg = donuts/static_cast<double>(milk);
        cout << donuts << " donuts.\n"
             << milk << " glasses of milk.\n"
             << "You have " << dpg
             << " donuts for each glass of milk.\n";
    }
    catch(NoMilk e)
    {
        cout << e.get_donuts() << " donuts, and No Milk!\n"
             << "Go buy some milk.\n";
    }
    cout << "End of program.";
    return 0;
}

NoMilk::NoMilk()
{}
NoMilk::NoMilk(int how_many) : count(how_many)
{}

int NoMilk::get_donuts()
{
    return count;
}
```

... Примеры диалогов такие же, как в листинге 17.2. ...

Обратите внимание на имеющийся в этой программе оператор throw:

```
throw NoMilk(donuts);
```

Здесь NoMilk(donuts) — вызов конструктора класса NoMilk. Конструктор принимает один аргумент типа int (в нашем случае значение переменной donuts) и создает объект класса NoMilk, который затем выбрасывается в качестве исключения.

Выброс и перехват исключений различных типов

Потенциально блок try может выбрасывать множество значений исключений, причем разных типов, но при каждом конкретном выполнении данного блока выбрасывается только одно из них (поскольку после этого выполнение блока try сразу же прекращается и у него просто не остается возможности выбросить еще одно исключение). Но в разных случаях во время выполнения блока try могут выбрасываться исключения разных типов. Хотя каждый блок catch может перехватывать значения только одного типа, в C++ существует возможность перехвата исключений разных типов — для этого после блока try следует поместить несколько блоков catch. Например, программа в листинге 17.4 содержит два таких блока.

Листинг 17.4. Перехват нескольких типов исключений

```
#include <iostream>     // Хотя в этом примере мы поступили иначе,
#include <string>       // классы исключения могут иметь собственные файлы интерфейса
using namespace std;    // и реализации и могут быть отнесены к пространствам имен.

class NegativeNumber    // Это лишь очередной пример.
{
public:
    NegativeNumber();
    NegativeNumber(string take_me_to_your_catch_block);
    string get_message();
private:
    string message;
};

class DivideByZero
{};

int main()
{
    int jem_hadar, klingons;
    double portion;

    try
    {
        cout << "Enter number of Jem Hadar warriors:\n";
        cin >> jem_hadar;
        if (jem_hadar < 0)
             throw NegativeNumber("Jem Hadar");

        cout << "How many Klingon warriors do you have?\n";
        cin >> klingons;
        if (klingons < 0)
```

продолжение ⇨

Листинг 17.4 (*продолжение*)

```
            throw NegativeNumber("Klingons");

        if (klingons != 0)
            portion = jem_hadar/static_cast<double>(klingons);
        else
            throw DivideByZero();
        cout << "Each Klingon must fight "
             << portion << " Jem Hadar.\n";
    }
    catch(NegativeNumber e)
    {
       cout << "Cannot have a negative number of "
            << e.get_message() << endl;
    }
    catch(DivideByZero)
    {
       cout << "Send for help.\n";
    }

cout << "End of program.\n";
    return 0;
}


NegativeNumber::NegativeNumber()
{}

NegativeNumber::NegativeNumber(string take_me_to_your_catch_block)
    : message(take_me_to_your_catch_block)
 {}

string NegativeNumber::get_message()
{
    return message;
}
```

Пример диалога 1

```
Enter number of Jem Hadar warriors:
1000
How many Klingon warriors do you have?
500
Each Klingon must fight 2.0 Jem Hadar.
End of program.
```

Пример диалога 2

```
Enter number of Jem Hadar warriors:
-10
Cannot have a negative number of Jem Hadar
End of program.
```

Пример диалога 3

```
Enter number of Jem Hadar warriors:
1000
How many Klingon warriors do you have?
0
```

```
Send for help.
End of program.
```

Обратите внимание, что здесь во втором блоке catch задан только тип параметра, DivideByZero, а его имя отсутствует. Если для обработки исключения значение параметра не требуется, можно задать только его тип. Этот случай подробнее описан в разделе «Совет программисту: классы исключений могут быть пустыми».

Ловушка: сначала перехватывайте специализированные исключения

Если программа перехватывает несколько типов исключений, порядок блоков catch может быть очень важен. Когда значение исключения выбрасывается в блоке try, блоки catch проверяются в порядке их следования в программе и выполняется первый блок, соответствующий типу этого исключения.

Вот блок catch особого вида:

```
catch (...)
{
    ... Здесь может быть любой код. ...
}
```

Он перехватывает исключения любого типа. Три точки не означают, что что-то пропущено, они должны быть введены в программе. Такой блок обычно помещают после всех остальных блоков, чтобы он выполнялся по умолчанию для исключений, типы которых не заданы в предшествующих блоках catch. Например, его можно добавить в программу листинга 17.4 так:

```
catch(NegativeNumber e)
{
   cout << "Cannot have a negative number of "
        << e.get_message() << endl;
}
catch(DivideByZero)
{
   cout << "Send for help.\n";
}
catch (...)
{
    cout << "Unexplained exception.\n";
}
```

Учтите, что этот блок всегда следует размещать после всех остальных блоков catch. В качестве примера предположим, что вместо приведенной выше последовательности мы расположили блоки следующим образом:

```
catch(NegativeNumber e)
{
   cout << "Cannot have a negative number of "
        << e.get_message() << endl;
}
catch (...)
{
```

```
    cout << "Unexplained exception.\n";
}
catch(DivideByZero)
{
   cout << "Send for help.\n";
}
```

В этом случае исключение типа NegativeNumber будет, как и предполагалось, перехватываться блоком NegativeNumber. А вот исключение типа DivideByZero будет перехватываться блоком catch (...), и блок catch для названного типа никогда не будет выполнен. К счастью, многие компиляторы сообщают о подобной ошибке.

Совет программисту: классы исключений могут быть пустыми

Рассмотрим определение класса DivideByZero из программы, приведенной в листинге 17.4.

```
class DivideByZero
{};
```

У данного класса нет переменных-членов и функций-членов (кроме используемого по умолчанию конструктора). Он имеет только имя, но этого вполне достаточно. Выброс объекта класса DivideByZero активизирует соответствующий блок catch, как в программе листинга 17.4.

При использовании пустого класса исключения в блоке catch объект такого класса обычно применяется только для того, чтобы войти в названный блок. Поэтому параметр блока можно не задавать. (Вообще говоря, параметр блока catch можно не задавать в любом блоке, независимо от того, является ли класс исключения пустым.)

Выброс исключений в функции

Иногда имеет смысл отложить обработку исключения. Например, можно написать функцию, выбрасывающую исключение при попытке деления на нуль, но не обрабатывающую его. Исключение будет перехватывать программа, использующая эту функцию, причем каждая такая программа может реагировать на него по-своему. Так, одни программы могут сразу завершать работу, а другие делать что-нибудь другое. В случаях, когда способ обработки исключения заранее неизвестен, внутри функции его перехватывать незачем. Функция, выбрасывающая и не обрабатывающая исключение, должна вызваться в блоке try, за которым следует блок catch.

Посмотрите на программу листинга 17.5. В ней есть блок try, но он не содержит оператор throw. Исключение в этой программе выбрасывает следующий оператор:

```
if (bottom == 0)
   throw DivideByZero();
```

однако непосредственно он находится вне блока try. Хотя программа как раз считает, что он располагается в этом блоке, входящем в состав функции main, но косвенно, поскольку данный оператор выполняется в функции safe_divide, вызываемой из блока try.

Листинг 17.5. Выброс исключения из функции

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

class DivideByZero
{};

double safe_divide(int top, int bottom) throw (DivideByZero);

int main()
{
    int numerator;
    int denominator;
    double quotient;
    cout << "Enter numerator:\n";
    cin >> numerator;
    cout << "Enter denominator:\n";
    cin >> denominator;

    try
    {
       quotient = safe_divide(numerator, denominator);
    }
    catch(DivideByZero)
    {
         cout << "Error: Division by zero!\n"
              << "Program aborting.\n";
         exit(0);
    }

    cout << numerator << "/" << denominator
         << " = " << quotient << endl;

    cout << "End of program.\n";
    return 0;
}

double safe_divide(int top, int bottom) throw (DivideByZero)
{
    if (bottom == 0)
        throw DivideByZero();

    return top/static_cast<double>(bottom);
}
```

Пример диалога 1

```
Enter numerator:
5
Enter denominator:
10
```

```
5/10 = 0.5
End of Program.
```

Пример диалога 2

```
Enter numerator:
5
Enter denominator:
0
Error: Division by zero!
Program aborting.
```

Спецификация исключений

Если функция не перехватывает исключение, она должна хотя бы предупреждать программиста, что ее вызов может выбрасывать исключение. Для этого выбрасываемые функцией типы исключений должны быть перечислены в спецификации исключений, как в следующем объявлении в программе листинга 17.5:

```
double safe_divide(int top, int bottom) throw (DivideByZero);
```

Отсюда видно, что спецификация исключений должна находиться и в объявлении функции, и в ее определении. И если у функции имеется более одного объявления, все они должны содержать одинаковые спецификации исключений (их иногда называют *списком* throw).

Если функция может выбрасывать исключения нескольких типов, эти типы разделяются запятыми, например:

```
void some_function() throw (DivideByZero, OtherException);
```

Все типы исключений, перечисленные в спецификации, интерпретируются обычным образом. В частности, функцию можно вызывать из блока try, за которым расположен блок catch, перехватывающий исключение того типа, который указан в списке throw данной функции. И когда функция выбросит исключение, этот блок catch его перехватит.

Если функция вообще не содержит спецификации исключений (списка throw), это не означает, что она не выбрасывает исключений, — напротив, это может говорить о том, что она может выбрасывать исключения любых типов, и они интерпретируются как обычно.

А вот когда функция содержит спецификацию, но выбрасывает исключение, не указанное в ней (и не перехватывает его сама), программа ведет себя иначе — в этом случае ее выполнение немедленно завершается. В частности, учтите, что выброшенное функцией исключение, не указанное в ее спецификации исключений, не будет перехватываться ни одним блоком catch. Помните, что полное отсутствие спецификации исключений эквивалентно списку всех возможных исключений. И в этом случае выполнение программы не завершается, а исключение обрабатывается как обычно.

Имейте также в виду, что спецификация исключений предназначена только для тех из них, которые не обрабатываются самой функцией и «выходят за ее пределы». Когда исключение не выходит за пределы функции (то есть обрабатывается в блоке catch внутри нее), его не нужно включать в спецификацию. Если определение функции содержит вызов другой функции, которая выбрасывает и не перехватывает

некоторое исключение, то его тип должен быть включен в спецификацию внешней функции.

Для указания того, что функция не должна выбрасывать исключений, которые она не перехватывает, следует задать пустую спецификацию исключений, вот так:

```
void some_function() throw ();
```

Теперь подведем итоги:

```
void some_function() throw (DivideByZero, OtherException);
// Исключения типа DivideByZero или OtherException
// интерпретируются как обычно. Все остальные исключения,
// не обработанные в теле функции, вызывают завершение
// работы программы.

void some_function() throw ();
// Пустой список исключений; все исключения, не обработанные
// в теле функции, вызывают завершение работы программы.

void some_function();
// Все исключения любых типов интерпретируются как обычно.
```

Не забывайте, что объект производного класса является объектом своего базового класса. Поэтому, если класс D производный от класса B, и класс B указан в спецификации исключения, выброс объекта класса D будет интерпретироваться как обычно, поскольку этот объект относится к классу B, указанному в спецификации. Однако на автоматическое приведение типов рассчитывать не следует. Например, если в спецификации исключений некоторой функции указан тип `double`, это не означает, что функция может выбрасывать исключения типа `int`. Для того чтобы она могла это делать и чтобы такие исключения обрабатывались программой, а не вызывали завершение ее работы, наряду с типом `double` нужно указать в спецификации исключений и тип `int`.

И последнее: не все компиляторы интерпретируют спецификации исключений так, как полагается, а некоторые из них вообще понимают их как комментарии, и тогда такие спецификации не оказывают никакого воздействия на программный код. Это еще одна причина, почему все исключения, которые могут выбрасываться функциями, нужно помещать в спецификацию. В этом случае все компиляторы будут интерпретировать исключения одинаково. Разумеется, такой же согласованности можно достичь, если вообще не использовать спецификацию исключений. Но тогда программа не будет так хорошо документирована и вы не сможете воспользоваться преимуществами дополнительного контроля ошибок, осуществляемого компиляторами, которые воспринимают спецификации исключений. Если компилятор обрабатывает спецификации исключений, то при появлении непредусмотренного исключения выполнение программы немедленно завершится.

Ловушка: спецификация исключений в производных классах

Если определение функции переопределяется или подменяется в производном классе, оно должно иметь такую же спецификацию исключений, как в базовом

классе, или же его спецификация исключений должна содержать подмножество типов исключений функции базового класса. Иными словами, переопределяя или подменяя определение функции, нельзя добавлять исключения в ее спецификацию (но при желании можно удалять некоторые из них). В этом есть смысл, поскольку объект производного класса не может использоваться везде, где может применяться объект базового класса, и поэтому его переопределенные или подмененные функции должны соответствовать коду, написанному для базового класса.

Упражнения для самопроверки

8. Что выводит следующая программа:

```
#include <iostream>
using namespace std;
void sample_function(double test) throw (int);
int main()
{
    try
    {
        cout << "Trying.\n";
        sample_function(98.6);
        cout << "Trying after call.\n";
    }
    catch(int)
    {
        cout << "Catching.\n";
    }
    cout << "End of program.\n";
    return 0;
}

void sample_function(double test) throw (int)
{
    cout << "Starting sample_function.\n";
    if (test < 100)
        throw 42;
}
```

9. Что выведет программа из предыдущего упражнения, если вызов

```
sample_function(98.6);
```

в блоке try заменить вызовом

```
sample_function(212);
```

17.2. Программирование обработки исключений

Пользоваться этим только в исключительных обстоятельствах.

Уоррен Пис

До сих пор мы говорили о том, как в C++ выбрасываются и перехватываются исключения, но пока не привели ни одного примера, демонстрирующего полезное

и реалистичное использование исключений. Теперь, когда вы освоили механизм обработки исключений, рассмотрим практические технологии.

Когда следует выбрасывать исключения

Выше в этой главе приведен очень простой код, демонстрирующий базовые концепции обработки исключений, однако этот код не реалистичен. Более сложным, но эффективным решением проблемы является разделение выброса и обработки исключений путем выполнения их в разных функциях. В большинстве случаев оператор throw следует выполнять в определении функции, указывая тип исключения в спецификации исключений этой функции, а ключевое слово catch помещать в другую функцию. Предпочтительная схема использования конструкции try...throw...catch такова:

```
void functionA() throw (MyException)
{
        .
        .
        .
    throw MyException(... Может присутствовать аргумент. ...);
        .
        .
        .
}
```

В другой функции (возможно, даже в другом файле):

```
void functionB()
{
        .
        .
        .
    try
    {
        .
        .
        .
        functionA();
        .
        .
        .
    }
    catch(MyException e)
    {
        ... Обработка исключения. ...
    }
        .
        .
        .
}
```

Более того, даже такой способ использования оператора throw должен применяться лишь в случаях, когда это неизбежно. Если проблему легко решить иным путем, лучше обойтись без выброса исключений. Пользуйтесь операторами throw тогда, когда способ обработки исключительной ситуации зависит от того, где

и как применяется данная функция. В таких случаях лучше всего предоставить обработку исключения программисту, вызвавшему функцию. Во всех остальных ситуациях желательно избегать выброса исключений.

Ловушка: необработанные исключения

Каждое выброшенное программой исключение должно быть перехвачено и обработано. Если этого не происходит, работа программы завершается.

> **Когда выбрасывать исключения**
>
> В большинстве случаев оператор `throw` должен использоваться в функции, а тип выбрасываемого им исключения задаваться в спецификации исключений этой функции. Более того, исключения следует выбрасывать только в тех случаях, когда способ обработки исключительной ситуации зависит от того, как и где используется функция. В таких случаях лучше всего предоставить обработку исключения программисту, вызвавшему функцию. Во всех остальных ситуациях желательно избегать выброса исключений.

Ловушка: вложенные блоки try...catch

Блок `try` и следующие за ним блоки `catch` можно поместить внутрь большего блока `try` или `catch`. Иногда это может быть полезно, но обычно существует лучший способ организации программы. Как правило, удобнее поместить внутренний блок `try...catch` внутрь определения функции, а ее вызов — во внешний блок `try` или `catch` (а может быть, и вовсе удалить один или несколько блоков `try`).

Если блок `try` и следующий за ним блок `catch` располагаются внутри большего блока `try` и во внутреннем блоке `try` выбрасывается исключение, не перехватываемое внутренним блоком `catch`, это исключение выбрасывается во внешний блок и может быть там перехвачено и обработано.

Ловушка: злоупотребление исключениями

Поток управления программ, в которых используются исключения, иногда может быть настолько запутан, что понять его крайне сложно. Выбросив исключения, можно перенаправить поток управления из одного места программы практически в любое другое. Когда-то давно, на заре эры программирования, такая неограниченная передача управления выполнялась с помощью оператора `goto`. Однако в настоящее время специалисты единодушны в том, что использование данной технологии является плохим стилем программирования. От этого оператора во многих языках давно отказались, а если он формально где-то и остался, им давно никто не пользуется. Исключения же позволяют снова вернуться к этой порочной практике, поэтому ими следует пользоваться крайне редко и только определенным образом. Лучше всего придерживаться такого правила: если вы считаете, что было бы неплохо воспользоваться оператором `throw`, сначала подумайте о том, как написать

программу или определение класса без него. И если найдется разумная альтернатива, воспользуйтесь ею, а не оператором throw.

Иерархия классов исключений

Очень полезно было бы определить иерархию классов исключений. Например, можно определить класс исключения ArithmeticError (арифметическая ошибка), а затем на его основе производный класс DivideByZeroError (деление на нуль). Поскольку объект класса DivideByZeroError одновременно является объектом класса ArithmeticError, каждый блок catch для класса ArithmeticError будет перехватывать и объекты класса DivideByZeroError. Если указать класс ArithmeticError в спецификации исключений, в нее автоматически будет включен и класс DivideByZeroError, независимо от того, зададите вы его явно или нет.

Проверка наличия свободной памяти

В главе 15 мы создавали новые динамические переменные с помощью следующего кода:

```
struct Node
{
    int data;
    Node *link;
};

typedef Node* NodePtr;
    ...
NodePtr pointer = new Node;
```

Это прекрасно работает до тех пор, пока для создания нового узла достаточно памяти. Но что произойдет, когда ее не хватит? В этом случае будет выброшено исключение bad_alloc. Тип bad_alloc является частью языка C++, и определять его не нужно.

Поскольку оператор new в случае нехватки памяти для создания нового узла выбрасывает исключение bad_alloc, в программе можно выполнять проверку наличия свободной памяти с помощью следующего кода:

```
try
{
    NodePtr pointer = new Node;
}

catch (bad_alloc)
{
    cout << "Ran out of memory!";
}
```

Конечно, вместо того чтобы сообщать пользователю о нехватке памяти, можно сделать что-нибудь более полезное, но это зависит от задач конкретной программы.

Повторный выброс исключения

C++ разрешает выбрасывать исключения из блока catch. Бывают случаи, когда, перехватив исключение и проанализировав его детали, можно выбросить это же или другое исключение для дальнейшей обработки цепочкой блоков catch.

Упражнения для самопроверки

10. Что произойдет, если исключение вообще не будет перехвачено?
11. Можно ли вкладывать один блок try в другой?

Резюме

- Обработка исключений позволяет проектировать и программировать нормальный ход работы программы отдельно от кода, обрабатывающего исключительные ситуации.
- Исключение можно выбросить в блоке try. В качестве альтернативы оно может быть выброшено из определения функции, не содержащего блока try (или не содержащего блока catch для перехвата данного типа исключения). В этом случае вызов функции следует поместить в блок try.
- Исключение перехватывается в блоке catch.
- За блоком try может следовать один или несколько блоков catch. При этом всегда сначала располагают блоки catch для специализированных классов исключений, а затем для более общих.
- Не злоупотребляйте исключениями.

Ответы к упражнениям для самопроверки

1. ```
   Try block entered.
   Exception thrown with
   wait_time equal to 46
   After catch block.
   ```

2. ```
   Try block entered.
   Leaving try block.
   After catch block.
   ```

3. ```
   throw wait_time;
   ```

   Обратите внимание, что следующий оператор является оператором if, а не throw, хотя и содержит оператор throw.

   ```
   if(wait_time > 30)
     throw wait_time
   ```

4. Когда выполняется оператор throw, выполнение блока try, в котором он находится, немедленно прекращается. Никакие другие операторы блока try не
   ```

выполняются, и управление передается одному из блоков catch, следующих за блоком try. Это означает, что выброшенное значение подставляется в параметр блока catch подходящего типа, а затем выполняется код этого блока.

5. ```
try
{
    cout << "Try block entered.";
    if (wait_time > 30)
        throw wait_time);
    cout << "Leaving try block.";
}
```

6. ```
catch(int thrown_value)
{
    cout << "Exception thrown with\n"
         << "wait_time equal to" << thrown_value << endl;
}
```

7. Параметром блока catch является thrown_value.

8. ```
Trying.
Starting sample_function.
Catching.
End of program.
```

9. ```
Trying.
Starting sample_function.
Trying after call.
End of program.
```

10. Если исключение нигде не перехватывается, программа завершается.

11. Да, блок try и соответствующие блоки catch могут располагаться внутри внешнего блока try. Однако лучше поместить внутренние блоки try и catch в определение функции и вызвать эту функцию из внешнего блока try.

Практические задания

1. Напишите программу, преобразующую значения времени из 24-часового в 12-часовой формат. Вот пример ее диалога:

```
Enter time in 24-hour notation:
13:07
That is the same as
1:07 PM
Again? (y/n)
y
Enter time in 24-hour notation:
10:15
That is the same as
10:15 AM
Again? (y/n)
y
Enter time in 24-hour notation:
10:65
There is no such time as 10:65
```

```
Try Again? (y/n):
Enter time in 24-hour notation:
16:05
That is the same as
4:05 PM
Again? (y/n)
n
End of program.
```

Вам нужно определить класс исключения TimeFormatMistake. Если пользователь введет несуществующее значение времени, например 10:65 или вообще что-то невразумительное вроде 8&*68, программа должна выбросить и перехватить исключение TimeFormatMistake.

2. Напишите программу, которая преобразует даты из числового формата месяц/день в текстовый формат месяц/день (например, 1/31 или 01/31 соответствует дате «January 31»). Диалог этой программы должен быть похож на диалог, приведенный в задании 1. Определите два класса исключений, MonthError (ошибка месяца) и DayError (ошибка дня). Если пользователь введет значение, отличное от существующего номера месяца (целые числа от 1 до 12), программа должна выбросить и перехватить исключение MonthError. Если же будет введено нечто отличное от существующего числа месяца (целые числа от 1 до 29, 30 или 31 в зависимости от номера месяца), программа должна выбросить и перехватить исключение DayError. Для простоты предполагается, что в феврале содержится 29 дней.

Приложение 1

Ключевые слова C++

Перечисленные ниже ключевые слова должны применяться только в соответствии с определенным им в языке C++ назначением. В частности, недопустимо их применение в качестве имен переменных или определяемых программистом функций. Помимо ключевых слов, приведенных в перечне, программы также не должны включать идентификаторы, начинающиеся двумя подряд идущими символами подчеркивания, — они зарезервированы для использования компилятором и стандартными библиотеками C++.

asm	do	Inline	return	typedef
auto	double	Int	short	typeid
bool	dynamic_cast	Log	signed	typename
break	else	Long	sizeof	union
case	enum	Mutable	static	unsigned
catch	explicit	Namespace	static_cast	using
char	extern	New	struct	virtual
class	false	Operator	switch	void
const	float	Private	template	volatile
const_cast	for	Protected	this	wchar_t
continue	friend	Public	throw	while
default	goto	Register	true	
delete	if	reinterpret_cast	try	

Следующие идентификаторы зарезервированы как альтернативные формы представления операторов и также не могут употребляться для других целей.

and &&	and_eq &=	bitand &	bitor |	compl ~	not !
not_eq !=	or ||	or_eq |=	xor ^	xor_eq ^=	

Приложение 2

Приоритет выполнения операторов

Все операторы, расположенные в одной ячейке приведенной ниже таблицы, имеют одинаковый приоритет. Чем выше в таблице расположена ячейка, тем более высоким приоритетом обладают находящиеся в ней операторы. Если приоритет унарных операторов и операторов присваивания одинаков, то они выполняются последовательно справа налево. Например, `x = y = z` означает `x = (y = z)`. Остальные операторы с равным приоритетом выполнятся слева направо. Так, `x + y + z` означает `(x + y) + z`.

Оператор	Назначение
`::`	Оператор расширения области видимости
`.`	Оператор точка
`->`	Оператор косвенного доступа к члену класса или структуры
`[]`	Индексация массива
`()`	Вызов функции
`++`	Постфиксный оператор инкрементирования (помещается после переменной)
`--`	Постфиксный оператор декрементирования (помещается после переменной)
`++`	Префиксный оператор инкрементирования (помещается перед переменной)
`--`	Префиксный оператор декрементирования (помещается перед переменной)
`!`	«НЕ»
`-`	Унарный минус
`+`	Унарный плюс
`*`	Оператор разыменования
`&`	Оператор взятия адреса
`new`	Динамическое выделение памяти
`delete`	Динамическое освобождение памяти
`delete[]`	Освобождение памяти, занимаемой динамическим массивом
`sizeof`	Получение размерности операнда в байтах

Оператор	Назначение
*	Умножение
/	Деление
%	Остаток (деление по модулю)
+	Сложение
-	Вычитание
<<	Оператор вывода
>>	Оператор ввода
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
==	Равно
!=	Не равно
&&	«И»
||	«ИЛИ»
=	Присваивание
+=	Сложение и присваивание
-=	Вычитание и присваивание
*=	Умножение и присваивание
/=	Деление и присваивание
%=	Остаток от деления и присваивание

Приложение 3

Набор символов ASCII

Приведены только печатные символы. Символом номер 32 является пробел.

32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	A	121	y
50	2	74	J	98	B	122	z
51	3	75	K	99	C	123	{
52	4	76	L	100	D	124	|
53	5	77	M	101	E	125	}
54	6	78	N	102	F	126	~
55	7	79	O	103	G		

Приложение 4

Некоторые библиотечные функции

Приведенный ниже перечень упорядочен по частоте использования функций, а не по их расположению в библиотеках. В определении функции приводятся типы ее аргументов (и их количество) и тип возвращаемого значения. В большинстве случаев объявления функций содержат только типы параметров без их имен (см. раздел «Альтернативная форма объявления функции» в главе 3).

Арифметические функции

Объявление функции	Описание	Заголовочный файл
`int abs(int);`	Абсолютное значение	`cstdlib`
`long labs(long);`	Абсолютное значение	`cstdlib`
`double fabs(double);`	Абсолютное значение	`cmath`
`double sqrt(double);`	Квадратный корень	`cmath`
`double pow(double, double);`	Возвращает первый аргумент в степени, заданной вторым аргументом	`cmath`
`double exp(double);`	Возвращает число e (основание натурального логарифма) в степени, заданной аргументом	`cmath`
`double log(double);`	Натуральный логарифм (ln)	`cmath`
`double log10(double);`	Логарифм по основанию 10	`cmath`
`double ceil(double);`	Возвращает наименьшее целое число, которое больше аргумента или равно ему	`cmath`
`double floor(double);`	Возвращает наибольшее целое число, которое меньше аргумента или равно ему	`cmath`

Функции-члены объектов ввода-вывода

Форма вызова функции	Описание	Заголовочный файл
имя_потока.open(*строка*)	Подсоединяет поток *имя_потока* к файлу, внешнее имя которого задается передаваемой в качестве аргумента строкой	fstream
имя_потока.fail();	Возвращает true, если при выполнении предшествующей операции над потоком *имя_потока* (например, open) произошла ошибка	fstream или iostream
имя_потока.close();	Отсоединяет поток *имя_потока* от файла, к которому он был подсоединен	fstream
имя_потока.bad();	Возвращает true, если поток *имя_потока* поврежден	fstream или iostream
имя_потока.eof();	Возвращает true, если программа попыталась выполнить чтение после последнего символа файла, к которому он подключен	fstream или iostream
имя_потока.get(*символьная_переменная*);	Считывает один символ из входного потока *имя_потока* и устанавливает значение своего аргумента равным этому символу. Не пропускает пробелы, символы табуляции и перевода строки	fstream или iostream
имя_потока.getline(*строковая_переменная*, *максимальная_длина* + 1);	Из потока *имя_потока* считывается одна строка и помещается в *имя_потока*. Если длина строки превышает значение *максимальная_длина*, считываются только первые символы, имеющие это значение. Переменная *имя_потока* должна иметь объявленный размер *максимальная_длина* +1 или больше	fstream или iostream

Форма вызова функции	Описание	Заголовочный файл
имя_потока.peek();	Считывает один символ из входного потока *имя_потока* и возвращает его. Но прочитанный символ не удаляется из входного потока. Следующая операция чтения прочитает этот же символ	fstream или iostream
имя_потока.put(*символ*);	Записывает символ в выходной поток *имя_потока*	fstream или iostream
имя_потока.putback(*символ*);	Помещает символ во входной поток *имя_потока*, чтобы следующая операция ввода прочитала это значение. Файл, к которому подключен входной поток, при этом не изменяется	fstream или iostream
имя_потока.precision(*целое_число*);	Задает количество знаков после десятичной точки для значений с плавающей запятой, выводимых в выходной поток *имя_потока*	fstream или iostream
имя_потока.width(*целое_число*);	Устанавливает ширину поля для следующего значения, выводимого в поток *имя_потока*	fstream или iostream
имя_потока.setf(*перечень_флагов*);	Устанавливает флаги форматирования для вывода в поток *имя_потока*. Список возможных флагов приведен в табл. 5.1	fstream или iostream
имя_потока.unsetf(*перечень_флагов*);	Сбрасывает флаги форматирования для вывода в поток *имя_потока*. Список возможных флагов приведен в табл. 5.1	fstream или iostream

Символьные функции

Настоящим типом аргументов всех этих функций является int, но в большинстве случаев его можно интерпретировать как char. Если функция возвращает значение типа int, для получения значения типа char необходимо явное или неявное приведение типа.

Объявление функции	Описание	Заголовочный файл
`bool isalnum(char);`	Возвращает true, если для данного аргумента функция isalpha или isdigit возвращает true. В противном случае возвращает false	cctype
`bool isalpha(char);`	Возвращает true, если аргумент является буквой. Может возвращать true и для других аргументов. Подробности зависят от реализации. В противном случае возвращает false	cctype
`bool isdigit(char);`	Возвращает true, если аргумент является цифрой. В противном случае возвращает false	cctype
`bool ispunct(char);`	Возвращает true, если аргумент является печатным символом, не удовлетворяющим условию функции isalnum (он не должен быть символом пробела, табуляции или перевода строки)	cctype
`bool isspace(char);`	Возвращает true, если аргумент является символом пробела, табуляции или перевода строки. В противном случае возвращает false	cctype
`bool iscntrl(char);`	Возвращает true, если аргумент является управляющим символом. В противном случае возвращает false	cctype
`bool islower(char);`	Возвращает true, если аргумент является буквой нижнего регистра. В противном случае возвращает false	cctype
`bool isupper(char);`	Возвращает true, если аргумент является буквой верхнего регистра. В противном случае возвращает false	cctype
`int tolower(char);`	Возвращает аргумент, переведенный в символы нижнего регистра. Если эквивалентного символа нижнего регистра не существует, возвращает аргумент без изменений	cctype
`int toupper(char);`	Возвращает аргумент, переведенный в символы верхнего регистра. Если эквивалентного символа верхнего регистра не существует, возвращает аргумент без изменений	cctype

Строковые функции

Объявление функции	Описание	Заголовочный файл
`int atoi(const char a[]);`	Преобразует символьную строку в целое число типа `int`	`cstdlib`
`long atol(const char a[]);`	Преобразует символьную строку в целое число типа `long`	`cstdlib`
`double atof(const char a[]);`	Преобразует символьную строку в число типа `double`	`cstdlib`[1]
`strcat(`*строковая_переменная, строка*`);`	Добавляет в конец строковой переменной указанную строку	`cstring`
`strcmp(`*строка1, строка2*`)`	Возвращает `true`, если заданные строки различаются между собой. В противном случае возвращает `false`[2]	`cstring`
`strcpy(`*строковая_переменная, строка*`);`	Заменяет значение строковой переменной указанной строкой	`cstring`
`strlen(`*строка*`)`	Возвращает длину строки	`cstring`
`strncat(`*строковая_переменная, строка, лимит*`);`	Такая же, как `strcat`, но добавляет максимум *лимит* символов	`cstring`
`strncmp(`*строка1, строка2, лимит*`)`	Такая же, как `strcmp`, но сравнивает максимум *лимит* символов	`cstring`
`strncpy(`*строковая_переменная, строка, лимит*`);`	Такая же, как `strcpy`, но копирует максимум *лимит* символов	`cstring`
`strstr(`*строка, подстрока*`)`	Возвращает указатель на первое вхождение заданной подстроки в строке. Если подстрока не найдена, возвращает указатель `NULL`	`cstring`
`strchr(строка, символ)`	Возвращает указатель на первое вхождение указанного символа в строке. Если символ не найден, возвращает указатель `NULL`	`cstring`
`strrchr(строка, символ)`	Возвращает указатель на последнее вхождение символа в строке. Если символ не найден, возвращает указатель `NULL`	`cstring`

[1] В некоторых реализациях C++ эта функция помещена в библиотеку `cmath`.

[2] Возвращает целочисленное значение, которое меньше нуля, равно нулю или больше нуля, в зависимости от того, больше `String_Exp1`, чем `String_Exp2`, равно ему или меньше него соответственно. Порядок сравнения строк лексикографический (сначала сравниваются первые два символа, и если они различны, то определяется результат сравнения; если же они равны, то сравниваются два следующих символа и т. д., пока одна из строк не закончится).

Генератор случайных чисел

Объявление функции	Описание	Заголовочный файл
`int random(int);`	Вызов функции `random(n)` возвращает псевдослучайное целое число, которое больше нуля или равно нулю и меньше n–1. (Имеется не во всех реализациях. Если этой функции нет, следует использовать `rand`)	`cstdlib`
`int rand();`	Вызов функции `rand()` возвращает псевдослучайное целое число, которое больше нуля или равно нулю и меньше `RAND_MAX`. `RAND_MAX`, – это предопределенная целочисленная константа из библиотеки `cstdlib`. Ее значение зависит от реализации, но оно не меньше 32 767	`cstdlib`
`void srand(unsigned int);` (`unsigned int` — это целочисленный тип, допускающий только неотрицательные значения. Можно считать его аргументом типа `int`, значения которого должны быть неотрицательными)	Повторно инициализирует генератор случайных чисел. Аргумент задает начальное число. После неоднократного вызова функции `srand` с одним и тем же аргументом функция `rand` или `random` генерирует одну и ту же последовательность псевдослучайных чисел. После вызова функции `rand` или `random` без предварительного вызова функции `srand` получается та же последовательность, что и после вызова последней с аргументом `1`	`cstdlib`

Тригонометрические функции

При работе с этими функциями нужно помнить, что значение угла задается в радианах, а не в градусах.

Объявление функции	Описание	Заголовочный файл
`double acos(double);`	Арккосинус	`cmath`
`double asin(double);`	Арксинус	`cmath`
`double atan(double);`	Арктангенс	`cmath`
`double cos(double);`	Косинус	`cmath`
`double cosh(double);`	Гиперболический косинус	`cmath`
`double sin(double);`	Синус	`cmath`
`double sinh(double);`	Гиперболический синус	`cmath`
`double tan(double);`	Тангенс	`cmath`
`double tanh(double);`	Гиперболический тангенс	`cmath`

Приложение 5

Макрос assert

Чтобы проверить, успешно ли был открыт файл in_stream, в главе 5 использовался следующий код:

```
if (in_stream.fail())
{
    cout << "Input file opening failed.\n";
    exit(1);
}
```

Эту же проверку можно выполнить с помощью макроса assert:

```
assert(!in_stream.fail());
```

Обратите внимание, что здесь применяется оператор отрицания, так как нас интересует, успешно ли завершилось открытие файла. Макрос assert принимает в качестве аргумента любое логическое выражение. Если результатом вычисления выражения окажется false, на экран будет выведено сообщение об ошибке и выполнение программы завершится. Если же результатом окажется true, программа выполнит оператор, следующий за макросом assert. Таким образом, макрос assert можно использовать для быстрой проверки ошибок в программе.

Этот макрос определен в библиотеке cassert (поскольку представляет собой конструкцию, подобную функции), и в текст любой программы, где он используется, необходимо включить следующую директиву include:

```
#include <cassert>
```

Преимуществом макроса assert является то, что его можно отключать. Обычно его применяют во время отладки программы, а затем отключают, чтобы пользователи не получали непонятных им сообщений об ошибках. Отключение макроса освобождает программу от лишней работы. Для отключения макроса assert в программу нужно добавить директиву #define NDEBUG перед директивой #include:

```
#define NDEBUG
#include <cassert>
```

Если в дальнейшем потребуется еще раз изменить программу и снова ее протестировать, макрос assert можно включить, удалив или закомментировав директиву #define NDEBUG.

Приложение 6

Встраиваемые функции

Если определение функции-члена очень короткое, его можно привести непосредственно в определении класса. Для этого следует просто заменить объявление функции-члена ее определением. Но в таком случае имя класса и оператор :: не задаются. Например, в следующем определении класса Pair определены два конструктора и функция-член get_first.

```
class Pair
{
public:
Pair()
    {}
    Pair(char first_value, char second_value)
        : first(first_value), second(second_value) {}
    char get_first()
    { return first;}
...
private:
    char first;
    char second;
};
```

Обратите внимание на то, что после закрывающей фигурной скобки в определении функции отсутствует точка с запятой, хотя ее можно поставить – это не будет ошибкой.

Функция, заданная в определении класса, называется *встраиваемой*. Определения встраиваемых функций обрабатываются компилятором не так, как определения обычных функций, поэтому они выполняются эффективнее, хотя и занимают больше места. При использовании такой функции каждый ее вызов в программе заменяется откомпилированной версией определения, так что ее вызовы в программе не приводят к издержкам, с которыми связаны вызовы обычных функций.

Приложение 7

Перегрузка квадратных скобок индекса массива

Оператор квадратные скобки, [], с помощью которого обычно выполняется обращение к элементам массива, можно перегрузить для применения с объектами класса. Если вы хотите пользоваться им в выражении, находящемся слева от оператора присваивания, определите его так, чтобы он возвращал ссылку. Для этого к имени возвращаемого типа добавьте символ &. (Как при перегрузке операторов ввода-вывода << и >>.) Перегружаемый оператор [] должен быть определен как функция-член; он не может быть дружественной функцией класса. (В этом отношении перегрузка оператора [] сходна с перегрузкой оператора присваивания. О ней, как вы помните, рассказывалось в разделе «Перегрузка оператора присваивания» главы 12.)

В качестве примера ниже приведено определение класса Pair, объекты которого ведут себя как массивы символов с индексами 1 и 2 (не 0 и 1):

```
class Pair
{
public:
    Pair();
    Pair(char first_value, char second_value);
    char& operator[](int index);
private:
    char first;
    char second;
};
```

Определение перегруженного оператора [] может быть таким:

```
char& Pair::operator[](int index)
{
    if (index == 1)
        return first;
    else if (index == 2)
        return second;
    else
```

```
    {
        cout << "Illegal index value.\n";
        exit(1);
    }
}
```

Объекты класса `Pair` объявляются и используются следующим образом:

```
Pair a;
a[1] = 'A';
a[2] = 'B';
cout << a[1] << a[2] << endl;
```

Заметьте, что в выражении `a[1]` идентификатор `a` представляет вызывающий объект, а `1` является аргументом перегруженного оператора `[]`.

Приложение 8

Указатель this

При определении функции-члена класса иногда нужно сослаться на вызывающий объект. Для этого можно воспользоваться специальным предопределенным указателем this. В качестве примера рассмотрим такой класс:

```
class Sample
{
public:
        ...
    void show_stuff();
        ...
private:
    int stuff;
        ...
};
```

Следующие два способа определения функции-члена show_stuff этого класса эквивалентны.

```
void Sample::show_stuff()
{
    cout << stuff;
}
// Не считайте это хорошим стилем программирования.
// Данный код просто демонстрирует использование указателя this.
void Sample::show_stuff()
{
    cout << (this->stuff);
}
```

Обратите внимание, что this — не имя вызывающего объекта, а имя указывающей на него переменной. Значение этого указателя не может быть изменено, он всегда указывает на вызывающий объект.

Как понятно из комментария, помещенного перед приведенным выше примером, указатель this редко нужен в программе. Однако в некоторых случаях он удобен и даже необходим.

Одна из ситуаций, в которых требуется использование указателя this, — перегрузка оператора присваивания =. Рассмотрим следующий класс:

```
class StringClass
{
public:
        ...
    StringClass& operator=(const StringClass& right_side);
        ...
private:
    char *a; // Динамический массив для строкового значения,
             // завершающегося символом '\0.'
};
```

Приведенное ниже определение перегруженного оператора присваивания может использоваться в операторах, подобных такому:

```
s1 = s2 = s3;
```

Эта цепочка операторов присваивания означает, что

```
s1 = (s2 = s3);
```

В следующем определении перегруженного оператора присваивания:

```
// Эта версия работает не для всех классов.
// Ниже приводится еще одна версия.
StringClass& StringClass::operator=(const StringClass& right_side)
{
    delete[] a;
    a = new char[strlen(right_side.a) + 1];
    strcpy(a, right_side.a);
    return *this;
}
```

указатель this применяется для получения объекта, заданного слева от оператора (то есть вызывающего объекта).

В большинстве случаев определение работает прекрасно, однако существует ситуация, когда оно действует неправильно. Если с двух сторон от оператора присваивания задан один и тот же объект (например, s = s;), член-массив вызывающего объекта удаляется. Чтобы этого не происходило, можно с помощью указателя this проверять, совпадают ли объекты, для которых вызван данный код:

```
// Окончательная версия с исправленной ошибкой:
StringClass& StringClass::operator=(const StringClass& right_side)
{
    if (this == &right_side)
    {
        return *this;
    }
    else
    {
        delete[] a;
        a = new char[strlen(right_side.a) + 1];
        strcpy(a, right_side.a);
        return *this;
    }
}
```

В разделе «Перегрузка оператора присваивания» главы 12 приводился пример перегрузки оператора присваивания для строкового класса StringVar. Там не требовался указатель this, поскольку класс содержал переменную-член max_length, с помощью которой можно было проверить, разные ли объекты заданы с двух сторон от оператора присваивания. Для описанного в данном приложении класса StringClass такой альтернативы нет, так как у него только одна переменная-член. Поэтому нам нужно воспользоваться указателем this.

Приложение 9

Перегрузка операторов как операторов-членов

В примерах этой книги мы обычно перегружали операторы как дружественные функции класса. Например, в листинге 8.5 из главы 8 приводился код дружественной функции, реализующей перегруженный оператор +. Оператор помечался как дружественный в определении класса:

```
// Класс для денежных сумм в валюте США.
class Money
{
public:
    friend Money operator +(const Money& amount1, const Money& amount2);
        ...
```

Далее мы определяли перегруженный оператор + вне определения класса (см. листинг 8.5).

Этот оператор + можно перегрузить и как оператор-член класса. Так же можно перегружать и любые другие операторы. Для перегрузки оператора + как члена класса нужно изменить определение данного класса следующим образом:

```
// Класс для денежных сумм в валюте США.
class Money
{
public:
    Money operator +(const Money& amount2);
```

Заметьте, что при перегрузке бинарного оператора как члена класса ему требуется всего один параметр. Вторым параметром является вызывающий объект. В качестве примера рассмотрим следующий код:

```
Money cost(1, 50), tax(0, 15), total;
total = cost + tax;
```

Когда оператор + перегружен как член класса, в выражении `cost+tax` переменная `cost` является вызывающим объектом, а переменная `tax` — аргументом оператора.

Определение оператора-члена + таково:

```
Money Money::operator +(const Money& amount2)
{
    Money temp;
    temp.all_cents = all_cents + amount2.all_cents;
    return temp;
}
```

Обратите внимание на следующую строку определения:

```
temp.all_cents = all_cents + amount2.all_cents;
```

Первым аргументом оператора является переменная all_cents, для которой явно не указано, к какому объекту она относится, следовательно, она является переменной-членом вызывающего объекта.

Перегрузка оператора как члена класса может показаться странной, но к ней легко привыкнуть. Многие специалисты настаивают, что операторы лучше перегружать именно так, поскольку это лучше согласуется с духом объектно-ориентированного программирования, чем перегрузка операторов как дружественных функций. Однако у названного способа имеется и один серьезный недостаток. При его использовании два аргумента перегружаемого оператора не симметричны. Один из них является вызывающим объектом, и только второй — «настоящим» аргументом, а это не только не эстетично, но и неудобно с практической точки зрения. Автоматическое приведение типов будет применяться только ко второму аргументу, поэтому, например, такой код:

```
Money base_amount(100, 60), full_amount;
full_amount = base_amount + 25;
```

является допустимым. Дело в том, что у класса Money имеется конструктор с одним аргументом типа long, и значение 25 рассматривается как значение этого типа, автоматически преобразуемое в значение типа Money.

Тем не менее при перегрузке оператора + как члена класса два его аргумента нельзя будет поменять местами. Следующее присваивание:

```
full_amount = 25 + base_amount;
```

будет недопустимым, поскольку 25 не может быть вызывающим объектом. Приведение значения типа long к типу Money выполняется для аргументов, но не для вызывающих объектов.

Если же перегрузить оператор + как дружественную функцию, присваивание, приведенное ниже, будет вполне допустимым:

```
full_amount = 25 + base_amount;
```

Алфавитный указатель

!

#define, директива, 427
#ifndef, директива, 427
#include, директива, 418
–, оператор вычитания, 73
--, оператор декрементирования, 90, 337
!, оператор «НЕ», 81
!=, оператор «не равно», 80
%, оператор, возвращающий остаток от деления, 74
&, оператор взятия адреса, 560
&&, оператор «И», 80, 82
*, оператор умножения и оператор разыменования, 73, 560
. (точка), оператор доступа к члену структуры и оператор доступа к члену класса, 198, 256
/, оператор деления, 73
||, оператор «ИЛИ», 81
+, оператор сложения, 73
++, оператор инкрементирования, 90, 337
<, оператор «меньше», 59
<<, оператор вывода, 37
<=, оператор «меньше или равно», 80
==, оператор «равно», 80
>, оператор «больше»
>=, оператор «больше или равно», 80
>>, оператор ввода, 37

A

abs, функция, 110, 384
assert, макрос, 759
atof, функция, 530
atoi, функция, 527
atol, функция, 527

B

bool, тип данных, 72, 310
break, оператор, 327, 347

C

C++
  история, 34
  компиляция и запуск программ, 41
  синтаксис ввода-вывода, 38
  структура программы, 39
  циклы, 335
case, ключевое слово, 327
catch, ключевое слово, 728, 731
ceil, функция, 110
char, тип данных, 70
cin, поток, 64
class, ключевое слово, 265
close, функция, 196
cmath, библиотека, 110
const, квалификатор, 97, 460
const, ключевое слово, 387
cout, поток, 59
cstdlib, библиотека, 527
cstring, библиотека, 520

D

default, ключевое слово, 328
delete, оператор, 565
do...while, оператор, 88, 335
do...while, цикл, 88, 335
double, тип данных, 67

E

eof, функция, 228
exit, функция, 200

F

fabs, функция, 110
fail, функция, 199
false, значение 80
float, тип данных, 69
floor, функция, 110
for, оператор, 340
friend, ключевое слово, 374
fstream, заголовочный файл, 194

G

get, функция, 220
getline, функция, 526, 537

I

if, оператор, 82
if...else, оператор, 78
  вложенный, 318
  многонаправленный, 321
ifstream, класс, 194
include, директива, 40, 61
int, тип данных, 67
iomanip, библиотека, 212
ios, флаг, 204
  ios::fixed, 210
  ios::left, 210
  ios::right, 210
  ios::scientific, 210
  ios::showpoint, 210
  ios::showpos, 210
iostream, заголовочный файл, 193
isalpha, функция, 233
isdigit, функция, 233
islower, функция, 233
isspace, функция, 234
isupper, функция, 233

L

labs, функция, 110, 384
long double, тип данных, 69
long int, тип данных, 69
long, тип данных, 69, 382

N

namespace, ключевое слово, 61, 139, 432
new, оператор, 561
null, константа, 659

O

ofstream, класс, 194
operator, ключевое слово, 392

P

pow, функция, 110
precision, функция, 209
private, ключевое слово, 272
protected, ключевое слово, 695, 696
public, ключевое слово, 274
put, функция, 220
putback, функция, 222

R

RAM, оперативная память, 23
return, оператор, 40, 117, 160

S

setf, функция, 209
setprecision, манипулятор, 212
setw, манипулятор, 212
size, функция-член вектора, 548
sqrt, функция, 110
std, пространство имен, 61, 430
stdlib, библиотека, 110
struct, ключевое слово, 225
strcat, функция, 521
strcmp, функция, 520

strcpy, функция, 519
string, библиотека, 532
  класс, 532
  функции-члены, 541
strlen, функция, 521
strncat, функция, 521
strncmp, функция, 521
strncpy, функция, 519, 521
switch, оператор, 326

T

this, предопределенный указатель, 763
throw, оператор, 729
tolower, функция, 233
toupper, функция, 233
true, значение, 80
try, ключевое слово, 728
try...throw...catch, конструкция, 732
typedef, оператор, 567

U

unsetf, функция, 211
using, директива, 61, 139, 430

V

vector, библиотека, 550
void, тип функций, 156, 192

W

while, оператор, 86, 335
while, цикл, 86
width, функция, 211

А

абстракция
  алгоритмическая, 638
  процедурная, 125
абстрактный тип данных, 293
автоматическое преобразование типов данных, 147
адрес памяти, 21
Айкен Говард, 44
алгоритм, 29
  последовательного просмотра, 479
алгоритм (*продолжение*)
  псевдокод, 128, 132
  разработка, 106
алгоритмическая абстракция, 638
Аль-Хорезми, 30
аппаратное обеспечение, 19
аргумент, 107
  механизм передачи, 137
  неверный порядок следования, 120
  строка, 522
  типа массива, 458, 462
  функции, 168
    используемый по умолчанию, 240
арифметический оператор, 73
арифметическое выражение, 73
ассемблер, 25

Б

байт, 21
бесконечная рекурсия, 605
бесконечный цикл, 93
безымянное пространство имен, 436
библиотека, 40
  cmath, 110
  cstdlib, 527
  cstring, 520
  fstream, 194
  iomanip, 212
  iostream, 40
  stdlib, 110
  string, 532
  vector, 550
  стандартных функций, 107
  функций, определяемая программистом, 429
бинарный оператор, 90
бит, 21
блок, 331
  видимость переменных, 334
  локальная переменная, 332
булево выражение, 80
Буль Джордж, 72
Бэббидж Чарльз, 28

В

ввод-вывод, 66
потоковый, 58
программирование, 66
с помощью класса string, 534
символьный, 219
синтаксис, 39
строк, 524
файловый, 193, 201
эхо-вывод, 66
вектор, 547
емкость, 550
размер, 548, 550
функция-член
capacity, 550
reserve, 551
size, 548
эффективное использование памяти, 550
ветвление
простое, 78
многонаправленное, 318
виртуальная функция, 705
вложенный оператор if...else, 318
выброс исключения, 729
вызов функции, 107
вызывающий объект, 198
выражение
арифметическое, 73
булево, 72, 80, 310
логическое, 72, 80, 310
сокращенное вычисление, 313

Г

граничное значение, 180

Д

данные, 24
двоичное число, 21
двухмерный массив, 495
декрементирование, 90
деструктор, 582
виртуальный, 716
деструктор (*продолжение*)
производные классы, 703
детализация пошаговая, 106
динамическая область памяти (куча), 565
динамический массив, 568
создание, 570
директива
#define, 427
#ifndef, 427
#include, 40, 61, 418
using, 61, 139, 430
область действия, 432
диск
гибкий, 23
жесткий, 23
компакт-диск, 23
дискета, 23
дружественная функция, 370, 374

Е

емкость вектора, 550

З

заглушка, 180
заголовочный файл, 418
cctype, 233
fstream, 194
iostream, 193
закрытый член класса, 269
зарезервированное слово, 53
зацикливание, 92
звуковой сигнал, 62
значение
true, 80
false, 80
возвращаемое функцией, 107
граничное, 180
переменной, 51
сравнение, 83
строковое, 516

И

идентификатор, 52
имя, 52

именованная константа, 96
имя идентификатора, 52
индекс элемента массива, 448
инициализация
  объекта, 284
  переменной, 57
  строковой переменной, 518
  структур, 261
инкапсуляция, 32, 266
инкрементирование, 90
информативное имя переменной, 58
исключение, 725
  выброс, 725, 729
  необработанное, 744
  обработка, 725
  обработчик, 730
  определение класса, 733
  повторный выброс, 746
  спецификация, 740
исключительная ситуация, 725
исполняемый оператор, 36
исходный код, 26
итератор, 668
итерация, 86

K

квалификатор const, 97, 387, 460
класс, 198, 264
  ifstream, 194
  ofstream, 194
  string, 532
    функции-члены, 541
  абстрактный тип данных, 293
  базовый, 683
  дочерний, 238, 683
  дружественная функция, 370, 374
  закрытый член, 269, 272
  интерфейс, 417
  инициализация, 284
  исключения, 733
  конструктор, 284
  многократно используемый, 426
класс (*продолжение*)
  наследование, 236
  открытый член, 269, 274
  переопределение функции-члена, 710
  подмена функции-члена, 710
  потомок, 690
  предок, 690
  производный, 236, 683
    деструктор, 703
  родительский, 238, 683
  свойства, 282
  унаследованный член, 688
  функция-член, 265
  член
    защищенный, 695
    ненаследуемый, 702
  шаблон, 644
ключевое слово
  struct, 255
  case, 327
  catch, 728, 731
  class, 265
  const, 387
  default, 328
  friend, 374
  namespace, 61, 139, 432
  operator, 392
  private, 272
  protected, 695
  public, 274
  try, 728
код
  исходный, 26
  машинный, 25
  объектный, 26
  программный, 26
комментарий
  однострочный, 94
  многострочный, 95
компилятор, 26
компиляция, 26, 41
  раздельная, 416

компоновка, 28, 416, 424
компоновщик, 26, 424
компьютер
  базовые классы, 19
  мэйнфрейм, 20
  основная память, 20
  персональный, 19
  рабочая станция, 20
  устройство ввода, 20
  устройство вывода, 20
константа
  ios, 204
  null, 659
  глобальная именованная, 135
  именованная, 96
  объявленная, 96
конструктор, 284
  автоматическое приведение типов, 395
  без аргументов, 291
  используемый по умолчанию, 289
  копирования, 585
  раздел инициализации, 288
  явный вызов, 289
конструкция try...throw...catch, 732

Л

Лавлейс Ада Августа, 28
лексикографический порядок (словарный), 520
локальная переменная, 133, 135, 332
логическое выражение, 80, 310
  вычисление, 310
  преобразование в тип int, 314

М

макрос assert, 759
манипулятор, 211
  setprecision, 212
  setw, 212
массив, 447
  адресация элементов, 450
  базовый тип, 448
  динамический, 568
массив (*продолжение*)
    создание, 570
    многомерный, 576
  для хранения строк, 515
  доступ к элементам, 447
  индекс элемента, 448
  инициализация, 453
  как член класса, 489
  как член структуры, 489
  многомерный, 493
  недопустимый индекс, 452
  объектов типа класса, 485
  объявление, 447
  передача в качестве аргумента, 458, 462
  поиск значения, 479
  размер, 448
  расположение в памяти, 450
  символов, 516
  сортировка, 481
  структур, 485
  частично заполненный, 475
  элемент, 448
машинный код, 26
механизм передачи параметров, 137
многонаправленное ветвление, 318
многонаправленный оператор if...else, 321
  вызов функции, 331
многострочный комментарий, 95
модификатор
  private, 272
  protected, 695
  public, 274
монитор, 20
мэйнфрейм, 20

Н

наследование, 32, 683
  классов, 236
нисходящее проектирование, 106
нуль-символ, 516
нуль-терминатор, 516

О

обеспечение
аппаратное, 19
программное, 24
обработчик исключений, 730
параметр, 730
объект, 192, 197
вызывающий, 198
инициализация, 284
типа string, 533
объектный код, 26
объявление
константы, 97
переменной, 36, 54
массива, 447
функции, 116
однострочный комментарий, 94
ОЗУ (оперативное запоминающее устройство), 23
ООП (объектно-ориентированное программирование), 32
оператор, 36, 761
–, 73
--, 90
!, 81
!=, 80
%, 74
&, 560
&&, 80, 82
*, 73
/, 73
||, 81
+, 73
++, 90
<, 59
<<, 37, 195
<=, 80
=, 55
==, 80
>, 80
>=, 80
оператор (*оператор*)
>>, 37, 195
перегрузка, 398
break, 327, 347
delete, 565
do...while, 88, 335
for, 340
if, 82
if...else, 78
вложенный, 318
многонаправленный, 321
return, 40, 117, 160
switch, 326
throw, 729
typedef, 567
while, 86, 335
арифметический, 73
бинарный, 90
ввода, 37
ветвления, 78
взятия адреса, 560
возврата, 40
вывода, 37, 59
вычитания, 73
декрементирования, 90, 337
деления, 73
доступа к члену класса по имени класса, 700
инкрементирования, 90, 337
исполняемый, 36
остатка от целочисленного деления, 74
составной, 84
перегрузка, 391
правила перегрузки, 395
присваивания, 55
перегрузка, 590
прямого доступа к члену класса, 198
пустой, 344
разыменования, 560
сложения, 73
составной, 84
точка, 256, 259

оператор (*оператор*)
умножения, 73
унарный, 90
целочисленного деления, 74
операторы
приоритет выполнения, 75
операция
арифметическая, 77
логическая, 80
определение функции, 116
открытый член класса, 269
отладка, 43
отладочная программа, 180
отступ, 94
очередь, 680
ошибка
логическая, 45
времени выполнения, 44
синтаксическая, 44

П

память
адрес, 21
байт, 21
бит, 21
вторичная, 20, 23
динамическая область (куча), 565
оперативная, 20
с последовательным доступом, 23
с произвольным доступом, 23
файл, 23
ячейка, 20, 46
параметр
выбор способа передачи, 169
константный, 389, 460
механизм передачи, 137
передача
по значению, 118
по ссылке, 162
смешанные списки, 168
строка, 522
типа массива, 457, 462
параметр (*продолжение*)
константный, 460
типа многомерного массива, 494
формальный, 116, 137, 168
выбор имени, 126
перегрузка
бинарных операторов, 394
имени функции, 143
оператора
ввода, 398
вывода, 398
правила, 395
присваивания, 590
унарных операторов, 396
основные понятия, 143
переопределение, 700
переменная, 50
автоматическая (обычная), 566
видимость во вложенных блоках, 334
глобальная, 136, 566
динамическая, 562, 566
значение, 51
идентификатор, 52
имя, 52, 58
индексированная, 448
инициализация, 57
локальная, 133, 135
для блока, 332
неинициализированная, 56
область видимости, 134
объявление, 36, 54
строковая, 516
инициализация, 518
тип данных, 54
трассировка, 357
указатель, объявление, 559
управления циклом for, 340
флаг, 352
переменная-объект, 194
переменная-член, 256
открытая, 269
закрытая, 269

переопределение и перегрузка, 700
переполнение стека, 609
перечисление, 317
подпрограмма, 107
полиморфизм, 32, 705
порядок лексикографический (словарный), 520
последовательность управляющая, 61
постусловие, 173
потерянный узел, 664
поток, 192
  cin, 64
  cout, 59
  ввода, 58
  входной, 193
    файловый, 194
  вывода, 58
  выходной, 193
    файловый, 194
  наследование, 732
  управления программы, 78, 310
правила
  перегрузки операторов, 395
  старшинства операций, 312
предусловие, 173
преобразование типов, 112
препроцессор, 110
приведение типов, 112
принцип
  процедурной абстракции, 125
  сокрытия информации, 125
  черного ящика, 125
приоритет выполнения операторов, 75
проблема расщепления, 712
проверка файла
  открытия, 199
  на конец, 212
программа, 19, 24
  виды ошибок, 44
  выполнение, 24
  главная часть, 40, 115
  запуск, 24, 27, 41
программа (*продолжение*)
  исходная, 26
  компиляция, 26, 41
  компоновка, 28
  объектная, 26
  отладка, 43
  отладочная, 180
  разработка
    этап реализации, 30
    этап решения задачи, 30
  тестирование, 179
  трансляция, 25
программирование
  объектно-ориентированное, 32
    инкапсуляция, 32
    наследование, 32
    полиморфизм, 32
  стиль, 93
программное обеспечение, 19, 24
  жизненный цикл, 33
программный код, 26
проектирование нисходящее, 106
простое ветвление, 78
пространство имен, 61, 217, 430, 431
  std, 61, 430
  безымянное, 436
  директива using, 139
  глобальное, 431
  выбор имени, 441
  область действия, 431
  создание, 432
прототип функции, 116
процедура, 107
процедурная абстракция данных, 173
процессор, 20, 24
псевдокод, 128, 132

Р

рабочая станция, 20
раздельная компиляция, 416
рекурсия, 598
  бесконечная, 605

рекурсия (*продолжение*)
перегрузка, 626
условие останова, 605, 611
Ричи Деннис, 34

С

связный список, 656, 661
вставка узла в начало, 662
вставка узла в середину, 669
поиск, 665
потерянный узел, 664
реализация стека, 673
удаление узла, 669
узел, 656
связывание
позднее (динамическое), 705
сеть, 20
сигнатура функции, 701
символ
двойная кавычка, 62
звукового сигнала, 62
новой строки, 37, 60, 62
нуль-символ, 516
обратная косая, 62
служебный, 61
табуляции, 62
управляющая последовательность, 62
символьная функция, 232
символьный ввод-вывод, 219
синтаксис
ввода-вывода, 38
оператора присваивания, сокращенный, 77
функции, 122
языка программирования, 55
синтаксическая ошибка, 44
система, операционная, 24
ситуация, исключительная, 725
слово
зарезервированное, 53
ключевое, 53
смешанный список параметров, 168
сокрытие информации, 124
сообщение об ошибке, 44
сортировка
методом выбора, 481
составной оператор, 84
спецификация исключений, 740
сравнение строк, 520
лексикографический (словарный) порядок, 520
станция, рабочая, 20
стек, 607
переполнение, 609
стиль программирования, 93
именованные константы, 96
комментарии, 94
отступы, 94
Страуструп Бьярн, 35
строка, 205, 515
ввод, 524
вывод, 524
использование операторов = и ==, 519
как аргумент, 522
как параметр, 522
новая, переход, 60, 62
обработка с помощью класса string, 539
перевод при вводе-выводе, 66
переход на новую, 37
преобразование в число, 527
присваивание, 519
строковая переменная, 516
инициализация, 518
строковое значение, 516
структура, 254
закрытые члены, 283
иерархическая (вложенная), 260
инициализация, 261
как аргумент функции, 259
переменная-член, 256
тег, 255
член, 256

Т

тело цикла, 86, 335
терминал, 20

тестирование программы, 179
тип данных, 54
bool, 72, 310
char, 70
преобразование в int, 384
double, 64, 67
float, 69
int, 67
long double, 69
long int, 69
long, 69, 382
short int, 70
short, 70
абстрактный, 293, 417
автоматическое преобразование, 147
имя, 54
определяемый программистом, 566
перечисление, 317
преобразование, 112
приведение, 112
к объектному типу, 395
символьный, 70
совместимость, 72
целочисленный, 69
числовой (общее описание), 69
Томпсон Кен, 34
точность числа, 70
трансляция, 25
трассировка переменной, 357

У

узел, 656
вставка в начало списка, 662
вставка в середину списка, 669
поиск в списке, 665
потерянный, 664
указатель, 558
арифметические операции, 574
как итератор, 668
зависший, 566
передача в функцию по значению, 584
переменная, 559
унарный оператор, 90
управление потоком, 77
управляющая последовательность символов, 62
условие, 80
останова (рекурсии), 611
устройство
ввода, 20
вывода, 20
утечка памяти, 665
уточнение имен, 434

Ф

файл, 23
заголовочный, 418
cctype, 233
cstring, 520
cmath, 10
fstream, 194
iomanip, 212
iostream, 193
stdlib, 110
string, 532
vector, 549
добавление данных, 204
приложения, 424
проверка
открытия, 199
на конец, 216, 228
файловый ввод-вывод, 193, 201
фактический параметр, 168
флаг, 209, 211, 352
ios, 209
ios::fixed, 210
ios::left, 210
ios::right, 210
ios::scientific, 210
ios::showpoint, 210
ios::showpos, 210
формальный параметр, 116, 168
выбор имени, 126
формат
выходных данных, 208

формат (*продолжение*)
представления чисел
инженерный, 68
с плавающей запятой, 68
с фиксированной запятой, 209
числа с дробной частью, 63
функция, 106, 199
abs, 110, 384
atof, 530
atoi, 527
atol, 527
ceil, 110
close, 196
eof, 228
exit, 200
fabs, 110
floor, 110
get, 220
getline, 526, 537
isalpha, 233
isdigit, 233
islower, 233
isspace, 234
isupper, 233
labs, 110, 384
main, 40
pow, 110
precision, 209
put, 220
putback, 222
seil, 110
setf, 209
sqrt, 110
strcat, 521
strcmp, 520
strcpy, 519
strlen, 521
strncat, 521
strncmp, 521
strncpy, 519, 521
tolower, 233
toupper, 233

функция (*продолжение*)
unsetf, 211
width, 211
аксессор, 275
аргумент, 107, 168
используемый по умолчанию, 240
типа массива, 458, 462
виртуальная, 705
возвращаемое значение, 107
возвращающая логическое значение, 316
встраиваемая, 760
вызов, 107, 116
вызов другой функции, 173
дружественная, 370, 374
заглушка, 180
заголовок, 117
итеративная, 609
логическая, 316
локальные переменные, 134
механизм передачи аргументов, 137
мутатор, 275
ненаследуемая, 702
область видимости переменных, 134
объявление, 116
определение, 114, 116, 122
определяемая программистом, 114
отладка, 180
параметр типа массива, 457, 462
параметр типа многомерного массива, 494
перегрузка имени, 143
передача параметров по значению, 118
передача параметров по ссылке, 162
переопределение, 710
подмена, 710
постусловие, 173
предусловие, 173
принцип процедурной абстракции, 125
принцип сокрытия информации, 124
принцип черного ящика, 125
прототип, 116
процедурная абстракция, 173

функция (*продолжение*)
расположение определений, 123
рекурсивная, 598
сигнатура, 701
символьная, 232
синтаксис, 122
создание, 115
стандартная, 107
тело, 117
тестирование, 182
типа void, 156, 192, 310
оператор return, 160
фактический параметр, 168
формальный параметр, 116, 158, 168
функция-член, 197, 265
переопределение, 690
рекурсивная, 623

X

Хопер Грейс Мюррей, 43

Ц

целочисленные типы данных, 69
цикл, 86, 335
do...while, 88, 335
for, 340
переменная управления, 340
while, 86, 335
использование фигурных скобок, 88
бесконечный, 92
завершение, 350
зацикливание, 92
использование, 344
цикл (*продолжение*)
итерация, 86, 335
тело, 86, 335
ЦПУ (центральное процессорное устройство), 20, 24

Ч

частично заполненный массив, 475
черный ящик, 124
число
двоичное, 21
с плавающей запятой, 68
с фиксированной запятой, 209
точность, 70
форматирование дробной части, 63

Ш

шаблон, 633
класса, 644
определение, 643

Э

элемент массива, 448
эхо-вывод, 66

Я

язык, машинный, 25
язык программирования
C, 34
ассемблер, 25
высокого уровня, 25
низкого уровня, 25
синтаксис, 55
ячейка памяти, 20, 46

Уолтер Савич

Программирование на C++

4-е издание

Перевод с английского О. Здир

Научный редактор	*В. Березовский*
Редактор	*Е. Насырова*
Художник	*Н. Биржаков*
Корректоры	*Е. Насырова, С. Езерницкая*
Технический редактор	*З. Лобач*

Лицензия ИД № 05784 от 07.09.01.

ООО «Питер Принт», 196105, Санкт-Петербург, ул. Благодатная, д. 67в.

ООО «Издательская группа BHV»
Свидетельство о занесении в Государственный реестр
серия ДК №175 от 13.09.2000.

Подписано к печати 06.11.03. Формат 70×100/16. Усл. п. л. 63,21.
Тираж 4500. Заказ 388
Налоговая льгота — общероссийский классификатор продукции
ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано с готовых диапозитивов
в ФГУП ордена Трудового Красного Знамени «Техническая книга»
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций
198005, Санкт-Петербург, Измайловский пр., 29.