П. Франка

Дискета прилагается



C++



учебный курс

▸▸ ПРОГРАММИРОВАНИЕ

Осваиваем самый популярный язык объектно-ориентированного программирования

ПИТЕР

Paulo Franca

C++

No experience required



П. Франка

C++



ПИТЕР®

Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Ростов-на-Дону • Екатеринбург • Самара
Киев • Харьков - Минск
2003

П. Франка

C++: учебный курс

Перевел с английского П. Бибиков

Главный редактор	*Е. Строганова*
Заведующий редакцией	*Я. Корнеев*
Литературный редактор	*А. Жданов*
Художественный редактор	*Я. Половодов*
Художник	*Н. Биржаков*
Корректор	*В. Листова*

ББК **32.973.2-018.1**

УДК 681.3.04

Франка П.

Ф83 C++: учебный курс. — СПб.: Питер, 2003. — **521** с.: ил.

ISBN 5–314–00136–5

Язык *C++* является в настоящее время одним из самых распространенных языков программировании, но одновременно и одним **из** самых трудных для изучения. Книга «C++: учебный курс» поможет быстро, эффективно и с наименьшими затратами освоить все основные приемы создания приложений на C++. Для изучения всех возможностей языка требуются объемные руководства и справочники, но эта книга даст вам «стартовый толчок», поможет понять структуру языка, принципы объектно-ориентированного программирования, методику проектирования и создания приложений. Учебный материал, содержащийся в 26-ти уроках, основан на практических примерах и сопровождается исходным кодом программ. Для его освоения не требуется никакой предварительной подготовки. Книга может быть использована в качестве учебного пособия для студентов, изучающих язык C++.

© Sybex, **1997**

© **Перевод** на русский язык, ЗАО Издательский дом «Питер», **2003**

© Издание на **русском языке,** оформление, ЗАО Издательский дом «Питер», 2003

Published by arrangement with the original publisher, Sybex Inc., U.S.A.
Подготовлено к печати ЗАО «Питер Бук» по лицензионному договору с Sybex Inc., США.

ISBN 5–314–00136–5
ISBN 0252–112111–X (англ.)

Все упомянутые в данном издании товарные знаки и зарегистрированные товарные знаки принадлежат своим законным владельцам.

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

ООО «Питер Принт». 196105, Санкт-Петербург, ул. Благодатная, д. 67в.
Лицензия ИД № 05784 от 07.09.01.

Подписано в печать 04.04.03. Формат 70×100$^{1}/_{16}$. Усл. п. л. 42,9. Доп. тираж 4500 экз. Заказ № 2728.

Налоговая льгота - общероссийский классификатор продукции ОК 005-93, том 2; 953005 - литература учебная.

Отпечатано с фотоформ в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое содержание

Часть 0. Подготовка к работе **22**
Урок 0. Установка программного обеспечения 23
Часть I. Ваши первые программы **82**
Урок 1. Разработка и модификация программ 83
Урок 2. Вывод информации на **экран** **94**
Урок 3. Решение проблем 107
Часть II. Функции и выражения 117
Урок 4. Функции 118
Урок 5. Числа 136
Урок 6. Решение проблем с помощью функций 150
Часть III. Циклы 170
Урок 7. Повторяющиеся вычисления 171
Урок 8. Разработка базовых циклов 195
Урок 9. Разработка базовых приложений 212
Часть IV. Условия **221**
Урок 10. Условные инструкции 222
Урок 11. Рекурсивные функции 248
Урок 12. Создание небольшого проекта 258

Часть V. Числа ... 266
Урок 13. Операции с числовыми выражениями ... 267
Урок 14. Работа с графикой ... 281
Урок 15. Создание анимаций ... 300
Часть VI. Классы ... 328
Урок 16. Создание и модификация классов ... 329
Урок 17. Производные классы ... 350
Урок 18. Усложнение приложений ... 366
Часть VII. Массивы и структуры ... 381
Урок 19. Массивы ... 382
Урок 20. Символьные массивы ... 407
Урок 21. Разработка приложений ... 424
Часть VIII. Ввод и вывод данных ... 436
Урок 22. Заголовочный файл franca.h ... 437
Урок 23. Файлы ... 458
Урок 24. Создание реального торгового терминала ... 484
Часть IX. Проблемы и решения ... 491
Урок 25. Создание объектно-ориентированного приложения ... 492
Урок 26. Дополнительные возможности ... 509
Алфавитный указатель ... 516

Содержание

Часть 0. Подготовка к работе 22

Урок 0. Установка программного обеспечения 23

Составление программ на C++ 25

Создание и выполнение программы 25

Зачем нужно компоновать программу **вместе с** другими? 25

Почему C++? **26**

Turbo **C++** и Visual **C++** **26**

Windows **3.1 или Windows 95?** **27**

Нужен ли вам компакт-диск? 27

Действительно ли нужно **писать** программы? 27

Зачем книге специальное **программное обеспечение?** **27**

Файлы и каталоги 28

Каталоги 28

Инсталляция и использование программного обеспечения 33

Инсталляция компилятора 34

Инсталляция программного **обеспечения книги** **34**

Запуск программ 35

Файлы **проекта** **36**

Различные компиляторы 37

Borland Turbo C++ 4.5 37

Borland C++ 4 44

Borland C++ 5 ..... **50**

Microsoft Visual C++ 1,5 56

Microsoft Visual C++ 4 64

Microsoft Visual C++ 5 ..... **73**

Звуковые файлы 80

Что нового мы узнали? 81

Часть I. Ваши первые программы ... 82
Урок 1. Разработка и модификация программ ... 83
Первая программа ... 84
Отправка сообщений ... 86
Инструкции ... 87
Класс athlete ... 89
Аргументы ... 89
Оформление программ на C++ ... 90
Понятие последовательности ... 91
Последовательные шаги ... 92
Инструкции Сэлу ... 92
Самостоятельная практика ... 93
Что нового мы узнали? ... 93
Урок 2. Вывод информации на экран ... 94
Объекты типа Clock ... 95
Сообщения объектам **типа** Clock ... 95
Использование **объектов** типа **Clock** ... 96
С чего начинается создание программы? ... 97
Добавление инструкций к комментариям ... 98
Объекты типа Box ... 99
Первый этап великого похода ... 100
Объекты типа Robot ... 100
Возможности робота ... 101
Перемещение робота ... 102
Разметка **трассы** ... 103
Повороты ... 103
Диалог **с роботом** ... 104
Лабиринт ... 104
Что нового мы узнали? ... 106
Урок 3. Решение проблем ... 107
Локализация проблемы ... 108
Пишите все что угодно ... 108
Дробление проблемы ... 109
Пользуйтесь готовыми решениями ... 109
Сообщения об ошибках .... .. ПО

Правила и соглашения 111
Идентификаторы 111
Ключевые слова 112
Типы и классы 112
Объявления 112
Сообщений 113
Комментарии 113
Самостоятельная практика 114
Поиск ошибки 114
Написание программ 114
Поиск ошибок 114
Правильные идентификаторы 115
Рисование 115
Поиск идентификаторов 116
Что нового мы узнали? 116
Часть II. Функции и выражения 117
Урок 4. Функции 118
Понятие функции 119
Создание функции 121
Аргументы функций 123
Аргументы и параметры 124
Значения и ссылки 127
Обучение объекта 129
Заголовочные файлы 130
Директивы 131
Самостоятельная практика 131
Область видимости 132
Примеры области видимости 134
Что нового мы узнали? **135**
Урок 5. Числа 136
Числа и числовые переменные 137
Общие правила для числовых переменных 139
Арифметические операторы в простых выражениях 141
Инкремент и декремент 142
Самостоятельная практика 143
Операторы отношения 143

Ввод значений....144
Функция ввода ask()....145
Имитация входного потока объектом On....145
Вывод значений....146
Имитация выходного потока объектом Cout....146
Что нового мы узнали?....149
Урок 6. Решение проблем с помощью функций....150
Возвращаемые значения функций....152
Типы возвращаемых значений....153
Встраиваемые функции....157
Второй этап великого похода....157
Использование функций....159
Правила и соглашения....162
Исключение лишних заголовочных файлов....165
Целые числа и числа с плавающей точкой....166
Выражения....167
Самостоятельная практика....168
Что нового мы узнали?....169
Часть III. Циклы....170
Урок 7. Повторяющиеся вычисления....171
Простые циклы....173
Цикл while....173
Самостоятельная практика....177
Цикл do...while....177
Цикл for....178
Условия в циклах....181
Проверка вводимых значений....181
Контроль времени....182
Функция yesno()....184
Вложенные циклы....186
Циклы и робот Tracer....188
Исследование комнаты....189
Самостоятельная практика....193
Что нового мы узнали?....194

Урок 8. Разработка базовых циклов 195
Повторение с изменением 196
Пример рисования квадратов разного размера 197
Преимущества циклов 199
Организация циклов 200
Самостоятельная практика 201
Первый этап создания циклов 201
Общие положения 201
Пример разработки цикла 203
Правила и соглашения 206
Составные инструкции 206
Вложенные инструкции 206
Область видимости 207
Простые условия 207
Самостоятельная практика 210
Что нового мы узнали? 211
Урок 9. Разработка базовых приложений 212
Создание торгового терминала 213
Программное обеспечение для нескольких транзакций 214
Полный листинг программы 217
Описание процесса разработки приложений 219
Пользовательский интерфейс 219
Исправления и модификации 220
Самостоятельная практика 220
Что нового мы узнали? 220
Часть IV. Условия 221
Урок 10. Условные инструкции 222
Инструкция if 223
Первый пример лабиринта 226
Самостоятельная практика 227
Первый пример магазина 228
Инструкция if...else 229
Самостоятельная практика 230
Второй пример лабиринта 230

Самостоятельная практика 231
Третий пример лабиринта 231
Второй пример магазина 233
Задание условий 234
Условные выражения 234
Составные выражения 235
Логические операторы 235
Вложенные условия 236
Выход из циклов 238
Инструкция break 239
Инструкция continue 240
Второй этап создания циклов 242
Выбор варианта 243
Пример телефонной станции 245
Что нового мы узнали? 247
Урок 11. Рекурсивные функции 248
Рекурсивные алгоритмы 249
Создание рекурсивных функций 250
Совершенствование алгоритма 253
Третий этап великого похода 255
Что нового мы узнали? 257
Урок 12. Создание небольшого проекта 258
Организация пользовательского интерфейса 259
Общее описание движений робота 259
Выбор вспомогательных функций 260
Контроль ошибок 262
Завершение проекта 263
Самостоятельная практика 264
Что нового мы узнали? 265
Часть V. Числа 266
Урок 13. Операции с числовыми выражениями 267
Обработка чисел 268
Идентификация числовых типов данных 268
Выражения 274
Возвращаемые значения функций 278
Библиотека math.h 279
Что нового мы узнали? 280

Урок 14. Работа с графикой 281
Графика 282
Размещение точек на экране 282
Изображение экранных объектов 283
Самостоятельная практика 290
Перемещение экранных объектов 290
Самостоятельная практика 293
Смена системы координат 293
Смена начала отсчета 293
Смена масштаба 294
Смена ориентации 294
Самостоятельная практика 296
Отказ от новых функций 296
Самостоятельная практика 297
Полярные координаты 297
Что нового мы узнали? 299
Урок 15. Создание анимаций 300
Построение графиков математических функций 301
Перемещение точки 302
Функции в качестве аргументов 305
Самостоятельная практика 306
Создание анимации 307
Иллюзия движения 307
Самостоятельная практика 312
Имитация пушечного ядра 313
Вычисление координат ядра 314
Моделирующий цикл 315
Самостоятельная практика 318
Одновременная обработка нескольких экранных объектов 318
Построение гимнаста 319
Класс Stage 321
Проект планетной системы 322
Объекты для проекта планетной системы 323
Моделирующий цикл 325
Самостоятельная практика 327
Что нового мы **узнали?** **327**

Часть VI. Классы 328
Урок 16. Создание и модификация классов 329
Создание классов 331
Составляющие класса объектов 332
Объявление класса объектов 335
Открытые, закрытые и защищенные члены класса 338
Конструкторы 339
Использование объектов 340
Создание нового класса объектов 341
Создание вагона с помощью конструктора 342
Создание вагона с помощью класса Stage 343
Доступ к членам класса 345
Открытый доступ к координатам 346
Конструктор с параметрами 348
Использование функции-члена 348
Самостоятельная практика 348
Что нового мы узнали? 349
Урок 17. Производные классы 350
Получение классов из существующих классов 351
Создание производных классов 352
Классы и объекты в реальном мире 353
Специализация экранных объектов 355
Наследование характеристик базовых классов 356
Несколько производных классов 358
Подмена унаследованной функции 361
Самостоятельная практика 364
Что нового мы узнали? 365
Урок 18. Усложнение приложений 366
Совершенствование торгового терминала 367
Выбор программного и пользовательского интерфейса 367
Интерфейс конечного пользователя 368
Объявление и определение класса 369
Программный код проекта 372
Проект имитации нескольких спутников 374
Разработка класса satellite 375
Реализация класса 377
Новая программа проекта планетной системы 378
Что нового мы узнали? 380

Часть VII. Массивы и структуры 381
Урок 19. Массивы 382
Работа с массивами 383
Варианты использования массивов 385
Число элементов массива 390
Самостоятельная практика 392
Числовые массивы 392
Использование массивов 395
Самостоятельная практика 396
Самостоятельная практика 397
Пример обработки числового массива 398
Поиск максимального элемента массива 401
Сортировка массивов 403
Обмен значениями 404
Рекурсивная сортировка 405
Что нового мы узнали? 406
Урок 20. Символьные массивы 407
Текст в программах на C++ 408
Тип char 408
Самостоятельная практика 410
Использование символьных массивов 410
Строковые функции 413
Преобразование символов в числа, и наоборот 416
Самостоятельная практика 417
Структуры 418
Структуры и массивы 419
Поиск в символьном массиве 421
Самостоятельная практика 423
Что нового мы узнали? 423
Урок 21. Разработка приложений 424
Проект торгового терминала 425
Реализация новых возможностей 425
Проект спутниковой системы 431
Расширение возможностей модели с помощью массивов 431
Самостоятельная практика 433
Что нового мы узнали? 435

Часть VIII. Ввод и вывод данных **436**

Урок **22.** Заголовочный файл franca.h 437

Реальное программирование на C++ 439

Проекты 440

Потоки ввода/вывода C++ 440

Заголовочный файл iostream.h 441

Потоковый вывод 442

Потоковый ввод 443

Ввод/вывод массивов 444

Форматирование 445

Точность значений с плавающей точкой 446

Выравнивание полей 446

Манипуляторы ввода/вывода 447

Выравнивание значений 451

Коды символов 454

Функции ask(), askwords() и yesno() 454

Самостоятельная практика 457

Что нового мы узнали? 457

Урок 23. Файлы 458

Использование файлов 459

Обработка файлов 460

Директивы препроцессора 468

Самостоятельная практика 471

Класс textfile для обработки файлов 471

Объявление класса 473

Данные-члены 473

Функции-члены 473

Конструкторы 474

Деструктор 475

Ввод с клавиатуры и вывод на экран 476

Считывание и запись 476

Поиск записи в файле 477

Самостоятельная практика 479

Базовые операции с файлами 479

Обновление данных в файле 479

Соответствие данных в файлах 480

Класс payfile 481

Самостоятельная практика 483
Что нового мы узнали? 483
Урок 24. Создание реального торгового терминала 484
Возвращение к проекту торгового терминала 485
Реализация новых возможностей 485
Поиск записей 489
Что нового мы узнали? 490
Часть IX. Проблемы и решения 491
Урок 25. Создание объектно-ориентированного приложения 492
Объектно-ориентированный менталитет 493
Переделка, доработка и многократное использование программ 493
Проект электронной игрушки 496
Предполагаемая реализация 496
Объектно-ориентированная реализация 498
Что нового мы узнали? 508
Урок 26. Дополнительные возможности 509
Дополнительные возможности C++ 510
Указатели 510
Комбинированные операторы 511
Перегрузка операторов 511
Шаблоны 512
Дружественные функции и классы 513
Множественное наследование 514
Выделение памяти 514
Логические операторы 514
Возможности программирования 514
Инженерные принципы в программировании 515
Что нового мы узнали? 515
Алфавитный указатель 516

Благодарности

С чего начать? Я думаю, что должен поблагодарить мой дорогой ноутбук! Сейчас ему уже три года — солидный возраст для компьютера. Он был со мной во многих аудиториях и просвечивался всеми видами рентгеновских лучей в разных аэропортах Южного и Северного Хемпшира. Не говоря уже о нескольких вирусных атаках! И все же он работал достаточно надежно, чтобы набрать на нем примерно 80 процентов этой книги.

Профессор Рамамурши-Рам (Ramamoorthy-Ram) своим воодушевлением первый побудил меня написать эту книгу. Профессор Дэниэл Левис (Daniel Lewis) очень помог мне на начальном этапе. Многие идеи, связанные с программным обеспечением и его представлением, родились из наших с ним дискуссий.

Я хочу поблагодарить также за постоянную поддержку Бразильский Исследовательский совет (Brazilian Research Council, CNPq) и федеральный университет Рио-де-Жанейро (Federal University of Rio de Janeiro, UFRJ). Этот университет, а особенно его компьютерный факультет и компьютерная лаборатория (NCE-UFRJ) оказали мне значительную поддержку. Я особенно благодарен профессору Адриано Крузу (Adriano Cruz), декану компьютерного факультета. Мне трудно выделить кого-то из сотрудников этого факультета, поскольку очень многие из них помогали мне в работе.

Спасибо всем, кто работает в издательстве Sybex, особенно Кристине Плэчи (Kristine Plachy) и издательской команде: Петер Кунц (Peter Kuhns), Рон Йост (Ronn Jost), Ким Вимпсетт (Kim Wimpsett) и Дейл Райт (Dale Wright), а также продюсерской команде: Эми Йофф (Amy Eoff), Робин Кибби (Robin Kibby) и Андре Бензи (Andrew Benzie).

Особая благодарность Андреа Франку (Andrea Franca) и Фредерико Арентцу (Frederico Arentz). Эта работа требовала от них продолжительных разъездов, на протяжении которых они, чтобы помочь мне, жертвовали многим в своей и без того занятой жизни.

Эни Ким (Eunhee Kim) играла особую роль. В каждом долгом предприятии бывают моменты усталости, когда хочется навсегда забросить работу. Но где бы мы ни были — рядом или за тысячи верст друг от друга, — она никогда не давала мне пасть духом. Благодаря ей я всегда был полон сил и уверен, что сумею довести дело до конца.

Санта Клара (Santa Clara) — это не просто школа, а действительно святое место — все это вдохновило меня начать и завершить эту работу.

Наконец, мой список будет неполным, если не упомянуть местные кафе и особенно непревзойденное Кофейное общество Купертино (Coffee Society in Cupertino), где была сделана значительная часть моей работы. Я глубоко признателен дружной и веселой компании, живущей в этих особенных местах. Я желаю им счастья и процветания.

Купертино, 24 июля 1997 г.

Введение

В давние-давние времена...

.....а на самом деле не так уж давно, изучение C++ и объектно-ориентированного программирования (Object Oriented Programming, OOP) было делом довольно трудным. Язык этот не слишком дружественен и вдобавок большая часть учебников по C++ была рассчитана на людей, уже владевших C.

Сторонники объектно-ориентированного программирования утверждают, и здесь я с ними полностью согласен, что начинающему лучше сразу усвоить объектно-ориентированную парадигму, чем изучать сначала традиционный подход, а затем переключаться на OOP. Кроме того, теперь уже ясно, что широкое использование C и все возрастающая популярность C++ в ближайшем будущем превратят C++ в основной язык программирования.

С другой стороны, я всегда интересовался тем, как подать учебный материал в наиболее интересной и оригинальной манере. Можно ли добиться этого в случае с C++ и объектно-ориентированным программированием?

В свое время, когда я задумал написать учебник, в университете Санта Клэру меня было несколько интересных встреч с главой отдела компьютерных разработок профессором Дэниэлом Левисом. Дэниэл отметил, что я должен либо создать что-нибудь революционное, либо вообще забыть об этом. Чтобы учить людей новой программной парадигме, нужна новая парадигма обучения.

Программное обеспечение книги

Именно новая парадигма обучения и дала толчок идее разработки специальных мультимедийных программ. Читателю не придется изучать все аспекты ввода/вывода только затем, чтобы видеть, как работают его программы. При наличии подходящей библиотеки классов для создания звука, картинок и анимации достаточно научиться манипулировать объектами этой библиотеки.

Программное обеспечение книги обеспечивает интерфейс с графической средой Windows и несколькими программными объектами, на которых читатель учится составлению программ. Эти объекты изображают людей, роботов, круги, квадраты, информационные рамки и т. д. Они существенно облегчают процесс обучения, поскольку с их помощью читатель может практиковаться сразу, даже не понимая всех сложных процедур ввода и вывода данных. Кроме того, благодаря этим объектам читатель может составлять алгоритмы, используя простые, интуитивно понятные и интересные геометрические примеры вместо обычных арифметических. Такое программное обеспечение настолько облегчает процесс обучения, что читать данную книгу без него просто не имеет смысла.

Я уверен, что создание вспомогательных программ, которые подходили бы для всех платформ, просто выше моих сил. Это как раз та цена, которую приходится платить за столь привлекательные средства обучения. Поэтому я ограничился разработкой программного обеспечения только для IBM-совместимых машин, поскольку они достаточно дешевы даже для студентов (в отличие от рабочих станций), и только для среды Microsoft Windows, поскольку это именно то направление, в котором развивается основное программное обеспечение для PC.

Кроме того, нужно было выбрать компилятор. Хотя начинал я с достаточно дешевого и удобного компилятора Borland Turbo C++, сейчас программное обеспечение книги работает и с последними версиями компиляторов Borland C++ и Microsoft Visual C++.

Создание книги вместе со всем необходимым программным обеспечением оказалось гораздо более трудоемким, чем я ожидал. Задача еще более усложнилась из-за необходимости программирования под Windows. Здесь я вынужден признаться, что не являюсь специалистом по Windows!

В любом случае дело сделано, и, я надеюсь, читатель останется доволен.



ПРИМЕЧАНИЕ Все исходные коды программного обеспечения книги находятся на прилагаемой дискете.

Что еще нового?

Кроме вспомогательного программного обеспечения в книге есть еще несколько необычных элементов. Она разбита на 10 частей, пронумерованных от 0 до IX. Почему от 0? Это поможет читателю усвоить, что в C++ индексы массивов всегда начинаются с нуля. Кроме того, материал части 0 носит в основном подготовительный характер. Здесь вы узнаете кое-что интересное о компьютерах и программистах, а также о том, как настраивать свои компиляторы (Turbo C++, Borland C++, Visual C++) для работы с предлагаемым программным обеспечением.



ПРИМЕЧАНИЕ Чтобы оценить свою готовность, перед тем как двигаться дальше, важно выполнить все упражнения части 0.

В частях I–III представлены базовые концепции программирования, и при этом читатель не перегружается синтаксическими деталями. Полнее о правилах синтаксиса можно узнать в конце каждой части. Это дает возможность читателю сначала понять назначение правил и только потом переходить к более сложным вопросам. Достаточно рано вводится понятие функции, что редко встречается в аналогичной литературе. Я испробовал этот подход на семинарах и был весьма удовлетворен результатами. Программистам, в первую очередь научившемся пользоваться функциями, обычно легче найти решение и разбить программу на функции, работать с

которыми гораздо удобнее. Помимо функций в самом начале книги читатель знакомится с объектами.

Одновременно необходимо сразу учиться составлять проекты, а не откладывать это до последней главы. Поэтому с первыми короткими проектами вы познакомитесь уже в части III. Один из этих проектов (торговый терминал) разрабатывается вместе с читателем на протяжении всей книги, начиная с совсем простой версии и заканчивая относительно сложным проектом, включающим каталоги, дисковые файлы и т. д.

Часть IV посвящена принятию решений и рекурсиям. К рекурсиям я также попытался подойти, используя графические и интуитивно понятные примеры.

Арифметические задачи специально отложены до части V. Простые выражения вводятся раньше, но реальные расчеты потребуются читателю только тогда, когда он будет готов сделать что-нибудь интересное. В части V все расчеты переносятся в геометрическую область. Студенты научатся программировать планеты, атомы, бильярдные шары, пушечные ядра и рисовать графики функций.

Объекты появляются уже на самом первом уроке. Однако создание и использование классов излагается только в части VI.

В части VII нетрадиционным способом определяются массивы. Сначала они вводятся как наборы экранных объектов (изображений людей, окружностей и т. д.) и только потом — как группы чисел и символов.

Как уже упоминалось, чтобы начать программировать, понимание ввода/вывода не обязательно. Читатель будет просто манипулировать появляющимися на экране объектами (анимациями), с которыми он знакомится, начиная с самого первого урока. Ввод/вывод через графический интерфейс Windows дается по мере надобности, реальные же аспекты ввода/вывода, включая файлы и форматирование, обсуждаются в части VIII.

В части IX даны основные приемы разработки программного обеспечения и представлен более сложный проект.

Любое понятие вводится только после того, как читатель готов к его восприятию. Некоторые понятия обсуждаются более чем в одной части, так что читатели могут изучать поведение того или иного компонента в разных ситуациях. Почти в каждой части имеется короткий проект, отражающий изученные опции разработки программ. Упражнения для проверки разбросаны по всему тексту под заголовком «Самостоятельная практика».

Дополнительные замечания, а также дополнительный материал для данной книги можно найти на Web-сайте издательства Sybex по адресу http://www.sybex.com.

Д-р Пауло Бианки Франка (Paulo Bianchi Franca, Ph.D.),

домашняя страница: http://www.franca.com.

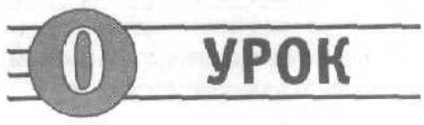
Часть 0

Подготовка к работе

В части 0 вы научитесь пользоваться компилятором C++ и программным обеспечением книги. Это краткое введение в программирование, а также знакомство с работой компилятора C++ и выполнением программ, разработанных специально, чтобы помочь вам легко и быстро освоить C++.

Материал книги предназначен для читателей, не имеющих навыков программирования. Однако некоторая компьютерная грамотность все же предполагается. Если вы умеете просматривать Web-страницы, загружать файлы и использовать электронную почту, то с программированием вы также должны справиться.

Хотя обучение с помощью специальных программ гораздо легче и удобнее, не следует слишком на них полагаться. На последних уроках, после освоения всего курса программирования на C++, вы узнаете, как обходиться без этих программ.



Установка программного обеспечения

- Задачи программиста
- **Файлы и каталоги**
- **Инсталляция и использование программ**
- **Специальные компиляторы**
- Запуск программ

Несмотря на распространенное мнение, программирование может быть довольно веселым занятием, дающим прекрасную возможность для тренировки ума и творческого воображения. Большинство людей, решивших заняться программированием, выбрали свою профессию именно потому, что им нравится программировать. Конечно, как и в любом деле, некоторые вещи интересны, а другие кажутся скучными. Но всегда делать только то, что нам нравится, невозможно.

Программировать на самом деле очень интересно. Вы чувствуете себя сильным, поскольку можете приказать компьютеру все, что захотите! Ваши возможности ограничиваются только вашими познаниями и вашим воображением.

Тем не менее учиться программированию интересно далеко не всегда. Кажется удивительным, но, используя компьютер для решения сложнейших задач математики, астрономии, физики, географии и т. д., в вопросах, связанных с обучением серьезному программированию, мы еще находимся где-то в меловом периоде!

Однако не отчаивайтесь! Не теряйте надежду! С помощью этой книги и созданных для нее программ (которые можно найти на прилагаемой к книге дискете) я постараюсь изложить предмет как можно интереснее и занимательнее, чтобы заинтересовать вас, пока энтузиазм еще не угас.

Сначала мне хочется дать один важный совет: работайте с компьютером! Чем больше времени вы проведете, сидя за ним, тем интереснее и эффективнее будет обучение. Если вы знаете, как составить программу другим способом, пожалуйста попытайтесь! Если у вас есть идеи, как сделать что-нибудь интересное, очень хорошо, ничто не должно вас останавливать.

Будьте настойчивы. Не просите вашего преподавателя показать вам то, что вы можете узнать сами!

Часть 0 содержит вводный материал, посвященный языкам программирования, файлам и т. д. Если вы все это уже знаете, то можете сразу начинать с раздела «Инсталляция и использование программного обеспечения».

ПРИМЕЧАНИЕ *Программирование* — это искусство давать точные указания для решения определенных задач, чтобы затем их можно было успешно выполнить без вашего участия. Конечно же, мы, как правило, будем иметь дело с компьютерными задачами, хотя и людям иногда приходится выполнять определенную последовательность действий, например, для приготовления по рецепту какого-нибудь блюда. При этом составитель рецепта должен дать очень подробное объяснение, чтобы повар затем мог действовать самостоятельно, не консультируясь с ним.

ПРИМЕЧАНИЕ Не начинайте урок 1, пока не убедитесь, что понимаете, как выполняется программа.

Составление программ на C++

Компьютеры пока не понимают нашего языка. Чтобы объяснить им наши требования, приходится использовать специальный язык— *язык программирования.* Компьютеры гораздо быстрее и надежнее людей, но уж никак не сообразительней! Для вас может показаться странным, что языки программирования гораздо проще обычных языков, но это так.

Тем не менее, хотя языки программирования проще естественных языков, компьютер даже их не в состоянии понять сам! Для работы ему нужен еще более простой *машинный язык.* К сожалению, этот язык настолько элементарен, что для человека он чрезвычайно скучен, а его использование сопряжено с большим количеством ошибок.

Создание и выполнение программы

С помощью специальных программ, называемых *компиляторами,* компьютеры могут транслировать (то есть переводить) с языка программирования на машинный язык. Компиляторы — это программы, переводящие на машинный язык инструкции, написанные на языке программирования.

Процесс создания программы включает несколько этапов:

1. Написание программы на некотором языке программирования, например на C++.
2. Трансляция программы с помощью компилятора в *объектный* файл (файл с расширением .OBJ).
3. Часто компьютер обнаруживает в программе ошибки и сообщает вам об этом. Тогда вам надо исправить свою программу и снова выполнить этап 2.
4. Компоновка вашей программы вместе с другими программами, которые могут потребоваться для ее выполнения. В результате вы получите свою программу в форме *исполняемого* файла (файл с расширением .EXE). Некоторые компиляторы выполняют этот этап автоматически.
5. Запуск программы. Часто из-за всевозможных логических ошибок ваша программа оказывается неработоспособной. В этом случае вам необходимо просмотреть и исправить ее, а затем повторить этапы 1-5,

Зачем нужно компоновать программу вместе с другими?

Некоторые программные задачи, хотя и кажутся простыми, на самом деле являются составными. Вывод предложения на экран компьютера, определение местопо-

ложения области, в которой был произведен щелчок мыши, и многие другие задачи выполняются за несколько этапов. Следовательно, написание программы для выполнения всех этих задач потребовало бы значительного времени.

Таким образом, чтобы ускорить и упростить процесс, в программу включаются некоторые предварительно созданные программные коды. Кроме того, чтобы помочь вам научиться программированию, я разработал ряд специальных программ, которые мы будем компоновать вместе с вашими программами.

Почему C++?

Сейчас используется много языков программирования. Все они (с помощью программ) позволяют объяснить компьютеру, что мы от него хотим. Язык C++ завоевывает все большую популярность, поскольку его возможности достаточно велики для создания даже самых сложных приложений.

Но C++ не детская забава! Он был разработан для профессиональных программистов, решающих сложные прикладные задачи, и не рассчитан на уровень новичка. Таким образом, C++ представляет собой определенный вызов как для вас (пытающихся научиться), так и для меня (пытающегося научить). Но все-таки я верю, что в конце концов вы тоже станете профессиональным программистом на C++! Благодаря моей методике вы сможете приступить к обучению безболезненно и, я надеюсь, даже с удовольствием[1].

Turbo C++ и Visual C++

Turbo C++ — это программное обеспечение, разработанное компанией Borland International, чтобы писать программы на C++. Компания Borland выпустила также другой продукт — Borland C++, — представляющий собой более полную (и дорогую) версию компилятора.

Visual C++ — это программный продукт, разработанный для тех же целей компанией Microsoft[2].

На рынке имеются и другие компиляторы C++.

ВНИМАНИЕ Компиляторы C++ не входят в программное обеспечение книги. Вам нужно купить один из компиляторов и установить его на своем компьютере.

[1] **После того как вы** пройдете курс **обучения языку C++, воспользуйтесь книгой** Дж. Элджера «C++: библиотека **программиста»** (издательство «Питер»). **Вы** найдете **в ней советы профессионала.**

[2] **О разработке** приложений в **среде** Visual C++ **читайте в книге С.** Холзнера «Visual C++ **6: учебный курс» (издательство** «Питер»).



СОВЕТ Посетите Web-страницу нашей книги. Возможно, после публикации этот список компиляторов пополнился.

Для работы с материалом этой книги вам подойдет любой из следующих компиляторов:

- Turbo C++ для Windows 4,5 (Borland).
- Borland C++ 4 или 5 для Windows.
- Microsoft Visual C++ 1.56, 4 или 5 для Windows.

Windows 3.1 или Windows 95?

Для запуска компилятора необходима операционная система. Ею должна быть либо Windows 3.1, либо Windows 95.

Конечно, в использовании разных компиляторов, равно как и в использовании той или иной операционной системы, имеются некоторые различия. На этом уроке вы узнаете, как вводить и запускать программы в каждом таком случае.

Нужен ли вам компакт-диск?

Совершенно не нужен! Многие до сих пор ассоциируют мультимедиа с компакт-диском. Мультимедиа подразумевает объединение средств анимации и звука. Для запуска мультимедийных приложений компакт-диск обычно необходим. Однако в этом курсе вы будете сами создавать свои мультимедийные программы, а не загружать их с компакт-диска.

Действительно ли нужно писать программы?

Вы можете научиться ездить на велосипеде, ни разу не садясь за руль? Я уверен, что нет. Точно так же нет иного способа научиться программированию, кроме как делать это на практике. Вам просто необходимо иметь доступ к компьютеру и достаточно времени для отладки программ.

Зачем книге специальное программное обеспечение?

Только так появляется возможность учиться с интересом и при этом не запоминать огромного количества разнообразной информации до появления реальных результатов. Если вы воспользуетесь составленными мной программами, то уже совсем скоро сможете работать над достаточно интересными задачами. Назначение программного обеспечения книги — сделать процесс обучения более приятным и захватывающим.

Файлы и каталоги

Я предлагаю вам несколько своих программ. Вы собираетесь создавать свои. Кроме того, некоторое количество разного рода программ уже имеется в вашем компьютере. Каждая такая программа хранится на жестком диске компьютера в виде *файла* со своим именем.

Обычно, но не всегда, имена файлов содержат *расширения,* то есть несколько дополнительных символов (не более трех), отделенных от основной части точкой.

Расширения определяют назначение файла. Например, программы, написанные на языке C++, имеют расширение .CPP.

Часто в этой книге мы будем использовать группы программ, составляющих так называемый *проект.* Проекты полезны тогда, когда вам необходимо скомпоновать несколько файлов, являющихся частями одной и той же программы. В частности, к книге прилагаются программы, которые вы будете компоновать в проекты совместно со своими программами.



ПРИМЕЧАНИЕ Использовать готовые части программ очень разумно, поскольку это существенно сокращает время создания новых программ, Основной целью объединения моих и ваших программ является создание более интересной среды разработки, которую вам не придется строить но пустом месте.

Каталоги

Поскольку компьютер различает файлы только по именам, важно, чтобы каждый файл имел свое имя. Чтобы упростить обращение к именам файлов, вы можете сгруппировать файлы в *каталог.* При создании любого файла помещайте его в свой каталог. Тогда, если вам потребуются ваши файлы, вы всегда найдете их в своем каталоге.

Если вы просите компьютер открыть файл с именем, которое уже имеет какой-либо другой файл, то все содержимое старого файла будет потеряно! Тем не менее можно иметь разные файлы с одинаковыми именами, если они находятся в разных каталогах.

Работа с каталогами

Для размещения программ, необходимых для работы с этой книгой, достаточно использовать один-единственный каталог. В нем вы будете создавать и хранить свои программы. Тем не менее при желании вы можете создавать любые дополнительные каталоги.

Все наши программы будут располагаться в каталоге, который мы назовем FRANCA (чтобы вы обо мне не забывали).

Так как этот каталог скорее всего будет расположен на диске C: вашего жесткого диска, то обращение к нему на экране вашего компьютера будет выглядеть следующим образом: C:\FRANCA,



ПРИМЕЧАНИЕ Вы можете разместить программное обеспечение в любом каталоге. Тем не менее в книге предполагается, что вы разместили его в каталоге C:\FRANCA.

Символ-заполнитель

Звездочка (*) означает произвольное количество любых символов. Ее можно использовать в операциях с именем файла. Такая звездочка обозначает любое возможное имя и называется также *символом-заполнителем*. Например, *.CPP означает все файлы с расширением .CPP, а *.* обозначает все возможные файлы.

Иерархия каталогов

Предположим, что вам понравилась идея каталогов и вы создали каталог с названием MYDIR, в котором разместили все свои файлы. Немного поработав, вы обнаруживаете, что в каталоге находится так много файлов, что в них можно запутаться.

Б такой ситуации имеет смысл размещать файлы в разных каталогах, то есть внутри каждого каталога создавать другие каталоги, называемые *подкаталогами*. Например, внутри каталога MYDIR вы можете создать три подкаталога COMPUT, PHISICS, PERSONAL.

Теперь у вас возникла *иерархиякаталогов*. Конечно, вы можете продолжить свою иерархию, создавая подкаталоги внутри уже созданных; практически здесь нет никаких ограничений, На рис. 0.1 показана возможная организация вашего жесткого диска. Как правило, ваш жесткий диск совпадает с диском C:. Вы можете организовать на диске C: несколько своих каталогов. На рисунке вы видите два таких каталога, это FRANCA и TCWIN. Внутри каталога FRANCA находится подкаталог OBJFILES. Хотя на рисунке показаны только эти каталоги, на вашем жестком диске обязательно имеются и некоторые другие.

Вы можете иметь одноименные файлы, расположенные в разных каталогах. Например, файлы с именем OHBOY.CPP могут находиться как в каталоге FRANCA, так и в каталоге TCWIN. Как компьютер различает такие файлы? Хороший вопрос. Но ведь вы каждый раз работаете в каком-то определенном каталоге (как будто вы выдвигаете только один ящик в своем шкафу). Каталог, в котором вы работаете в настоящий момент, называется *текущим каталогом*. Компьютер всегда предполагает, что вы работаете с файлами текущего каталога. Однако иногда вы можете обозначать файл полностью, используя иерархию каталогов. Это называется *полностью определенным именем файла* или *полным путем к файлу*.

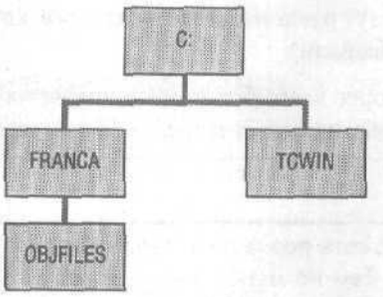


Рис. 0.1. *Иерархиякаталогов*

В нашем примере полные пути к файлу имеют следующий вид:

```
C:\FRANCA\OHBOY.CPP
C:\TCWIN\OHBOY.CPP
```

Двигаемся по иерархии каталогов

Допустим, вы хотите открыть файл внутри вашего каталога PERSONAL. Запомните, что этот каталог находится внутри каталога MYDIR. Если вы попробуете открыть этот файл в компиляторе типа Turbo C++4.5, то увидите на экране нечто похожее на рис. 0.2, Если вы пользуетесь компилятором Windows 95, то экран будет напоминать рис. 0.3.

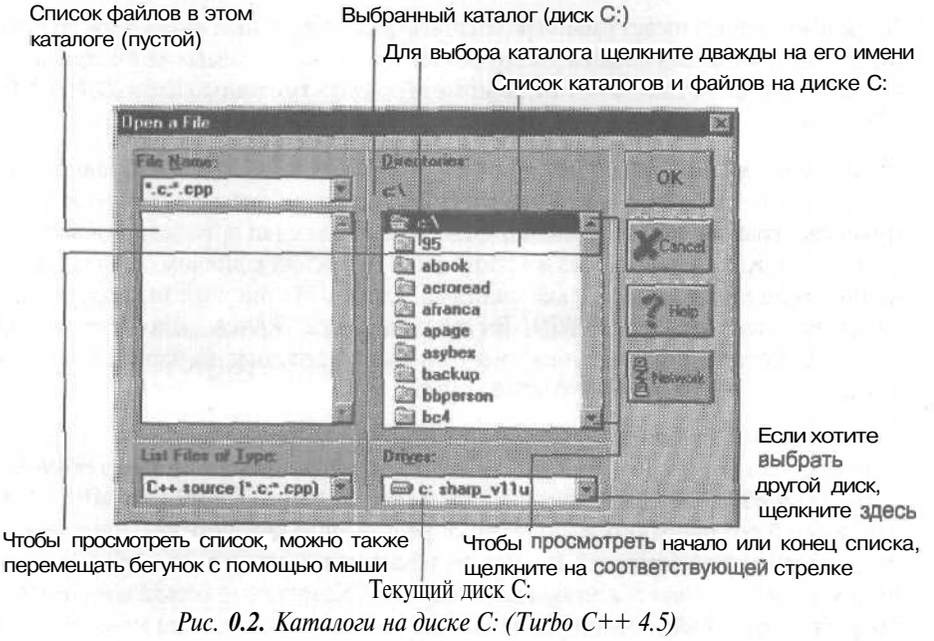


Рис. ***0.2.*** *Каталоги на диске C: (Turbo C++ 4.5)*

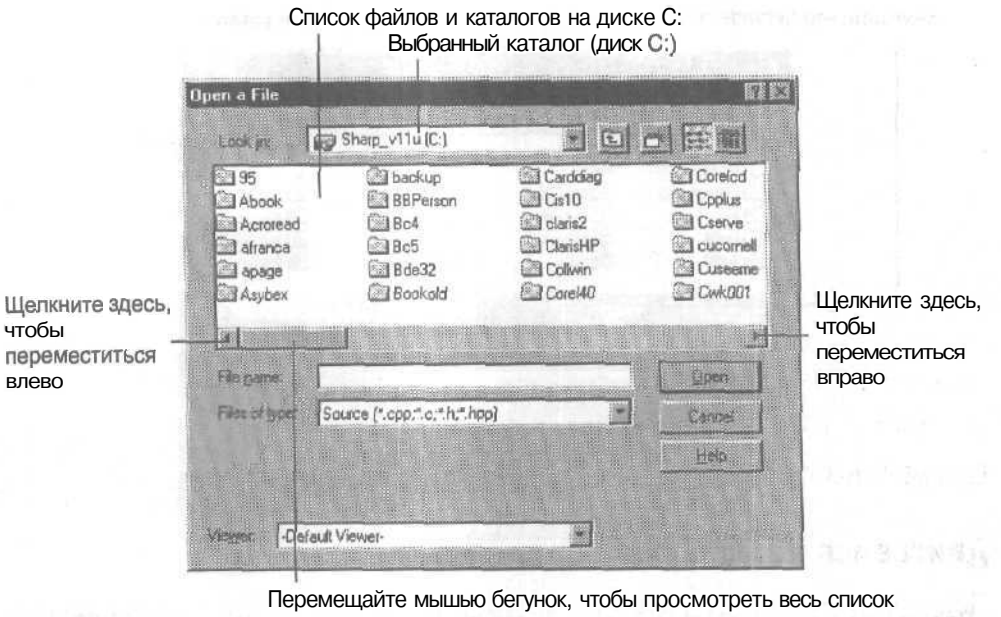


Рис. 0.3. *Каталоги на диске C: (Borland C++ 5)*

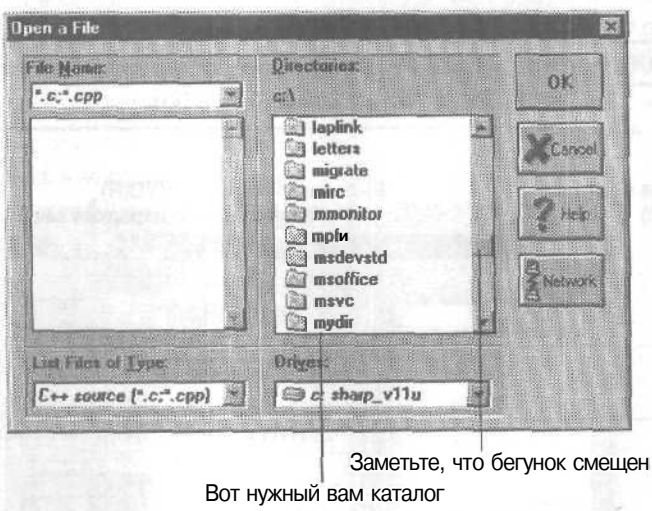


Рис. 0.4. *Поиск каталога MYDIR (Turbo C++ 4.5)*

В этом примере, кроме каталога MYDIR, на диске имеется много других каталогов (как, наверно, и на вашем компьютере). Так что полный список может даже не поместиться на экране. Что же делать? Воспользуйтесь бегунком или стрелкой, чтобы прокрутить список, пока не появится нужный вам каталог. Тогда список будет выглядеть так, как показано на рис. 0.4 или рис. 0.5.

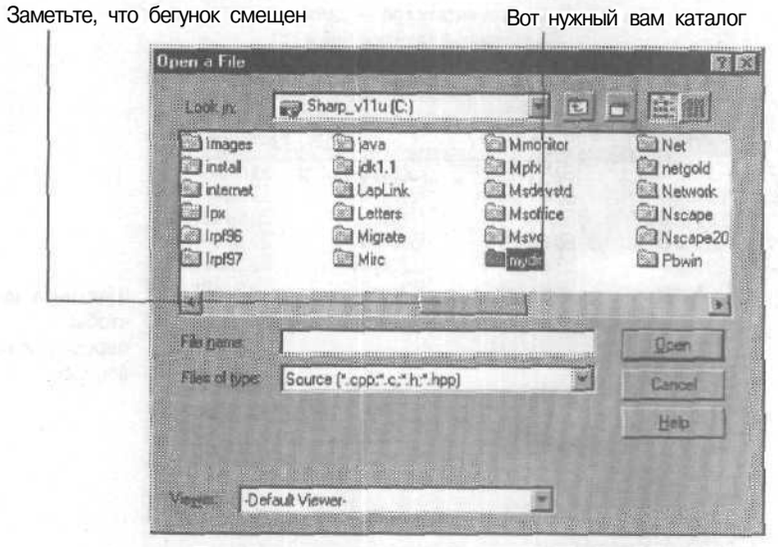


Рис. 0.5. *Поиск каталога MYDIR (Borland C++ 5)*

Все, что вам нужно сделать теперь, — это щелкнуть два раза на каталоге MYDIR. Вы увидите новый экран, похожий либо на рис. 0.6, либо на рис. 0.7.

ПРИМЕЧАНИЕ Это фиктивный пример. Предполагается, что у вас имеется каталог MYDIR с подкаталогом PERSONAL.

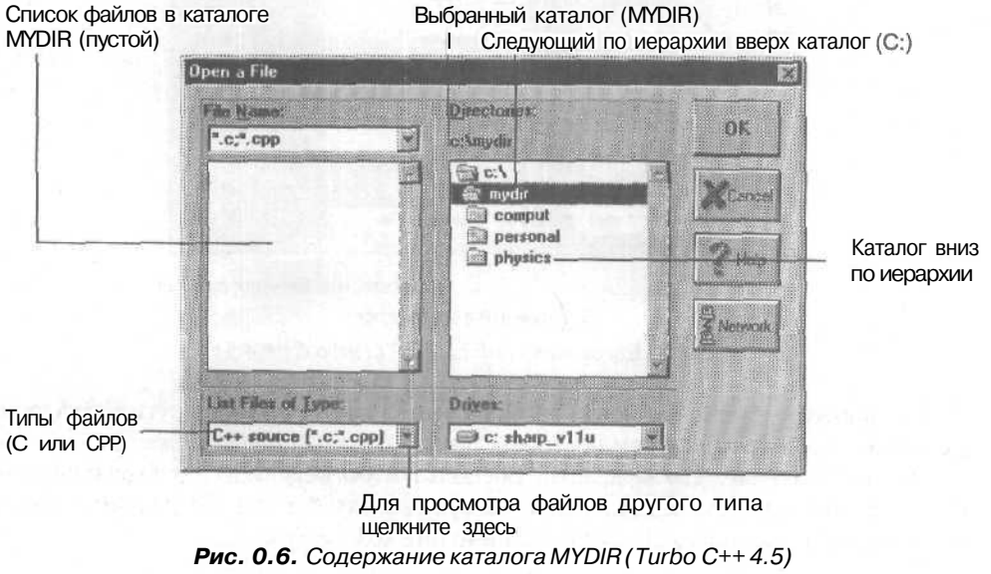


Рис. 0.6. *Содержание каталога MYDIR (Turbo C++ 4.5)*

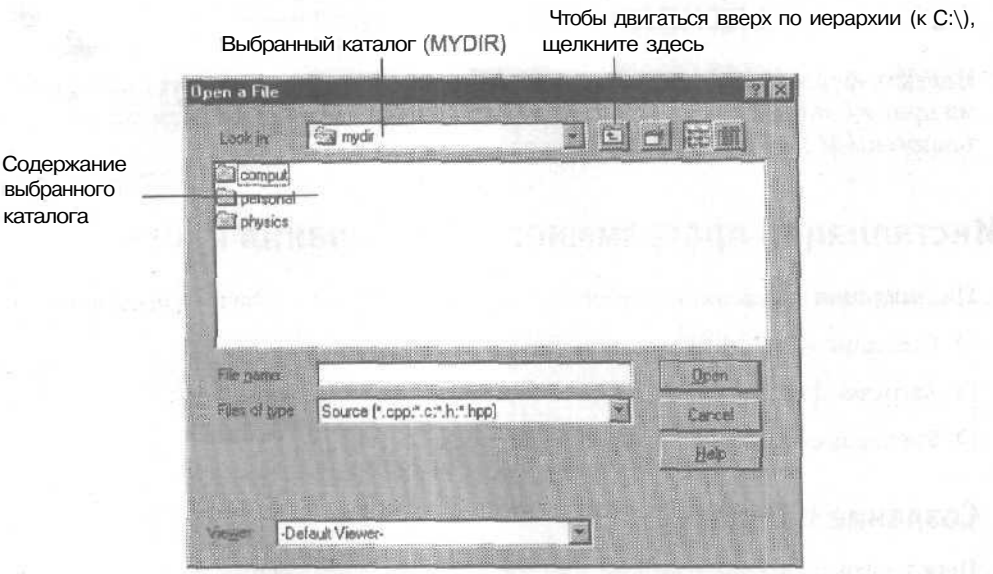


Рис. 0.7. *Содержание каталога MYDIR (Borland C++ 5)*

Копии программ

По мере того как вы будете создавать свои или модифицировать мои программы, вам понадобится их сохранять, причем так, чтобы иметь возможность в любую минуту просмотреть или использовать любую из них.

Не назначайте программам уже существующие в вашем каталоге имена. Когда вы просматриваете или модифицируете одну из моих программ, старайтесь сохранять ее под другим именем (или вообще не сохраняйте); иначе вы потеряете исходную программу. Если же это все-таки случилось, а исходная программа вам все-таки нужна, вы всегда сможете переписать ее с прилагаемой к книге дискеты.

Создание копий ваших программ — это хорошая практика программирования. Иначе вы можете случайно стереть программу, и вся работа пойдет насмарку.

Инсталляция и использование программного обеспечения

Цель этой книги — научить вас программированию и использованию компьютера для решения ваших задач. Что касается компиляторов, то здесь рассматривается только наиболее существенная информация. Если вам понадобится дополнительный материал, воспользуйтесь системой контекстной помощи соответствующего компилятора.

Инсталляция компилятора

Инсталлируйте компилятор (Borland или Microsoft) в соответствии с инструкциями производителя. Очень важно, чтобы вы знали и помнили, в каком каталоге установлены файлы вашего компилятора.

Инсталляция программного обеспечения книги

Инсталляция программного обеспечения книги состоит из следующих этапов:

- Создание каталога на жестком диске.
- Загрузка файлов.
- Распаковка файлов.

Создание каталога

Перед загрузкой программного обеспечения вам необходимо создать каталог на своем жестком диске. Я рекомендую создать каталог FRANCA в корневом каталоге диска C:.

Это можно сделать с помощью проводника Windows (Windows Explorer) или диспетчера файлов (File Manager) Windows 3.1.

В окне проводника Windows щелкните один раз на диске C:, а затем выберите в меню Файл (File) команду Создать (New) и далее команду Папка (Folder), наберите имя FRANCA и нажмите кнопку OK.

Загрузка программного обеспечения

Скопируйте файл CPPNER.EXE с прилагаемой к книге дискеты в каталог FRANCA.

Распаковка файлов

Все программное обеспечение находится в упакованном виде, так что вы можете загрузить все одновременно и сэкономить время при загрузке. Файл CPPNER.EXE — это самораспаковывающийся архив, который содержит все необходимые исходные файлы.

Распаковать файлы можно в проводнике Windows, для этого просто дважды щелкните на файле CPPNER.EXE и в появившемся диалоговом окне введите путь и имя каталога, после чего в вашем каталоге C:\FRANCA появится несколько файлов и подкаталог sounds (звуки).

Теперь все готово к работе. Файл CCPNER.EXE в принципе можно стереть, но лучше сохранить его на случай, если вам придется восстанавливать свои файлы.

На протяжении всей книги вы будете встречать ссылки на программы, установленные в каталоге FRANCA на диске C:. Вы можете, если хотите, скопировать эти файлы в другой каталог.



ПРИМЕЧАНИЕ Если программа находится на сетевом устройстве, доступном только для чтения, то может потребоваться скопировать ее на устройство, доступное как для чтения, так и для записи. Файл проекта *должен* быть размещен на устройстве для чтения и записи, равно как и те программы, которые вы будете создавать. То же самое относится к объектным файлам и файлам ресурсов,

Запуск программ

Для запуска программ из этой книги вам понадобятся компилятор и программное обеспечение. Способы работы с разными компиляторами несколько отличаются. Более подробные рекомендации даны в следующем разделе.



ВНИМАНИЕ Для каждой среды разработки нужен свой файл проекта. В противном случае существует вероятность, что программы не запустятся.

Суть операций, выполняемых компилятором:

- Запуск компилятора.
  - Чтобы ваши программы работали, необходимо запустить компилятор.
- Открытие, редактирование и закрытие проекта.
  - Проект — это набор программ, связанных между собой и выполняемых вместе. Чаще всего придется объединять ваши программы с программами из каталога C:\FRANCA.
  - Большинство программ, которые вам придется выполнять, являются частями проекта. Вы должны знать, как выбрать проект, с которым будете работать. Это называется *открытием проекта*.
  - Вообще говоря, ваши проекты будут состоять из трех файлов: PAULO.RC, FRAN***.OBJ (символы, скрытые звездочками, зависят от типа вашего компилятора) и исходной программы (сначала это будет программа C_SAL.CPP).
  - Поскольку проект — это набор программ, вам часто придется удалять из проекта или добавлять к нему те или иные программы. Используя все время один и тот же проект, вы будете менять в нем соответствующие исходные программы.

- Открытие, редактирование и закрытие файла программы.
  - Проект содержит файлы программ. Часть из них остается неизменной, а другие вы перемещаете и совершенствуете. Вы должны уметь просматривать и модифицировать исходные программы. Вы также должны уметь сохранять эти программы.
- Выполнение проекта.
  - Чтобы увидеть работу программы, вам нужно выполнить проект. Выполнение проекта предполагает выполнение всех его составляющих. В случае успеха вы увидите на экране компьютера результаты выполнения программы.
  - Если вы выполните проект с файлом C_SAL.CPP, то ваш экран будет выглядеть примерно так, как показано на рис. 0.8.

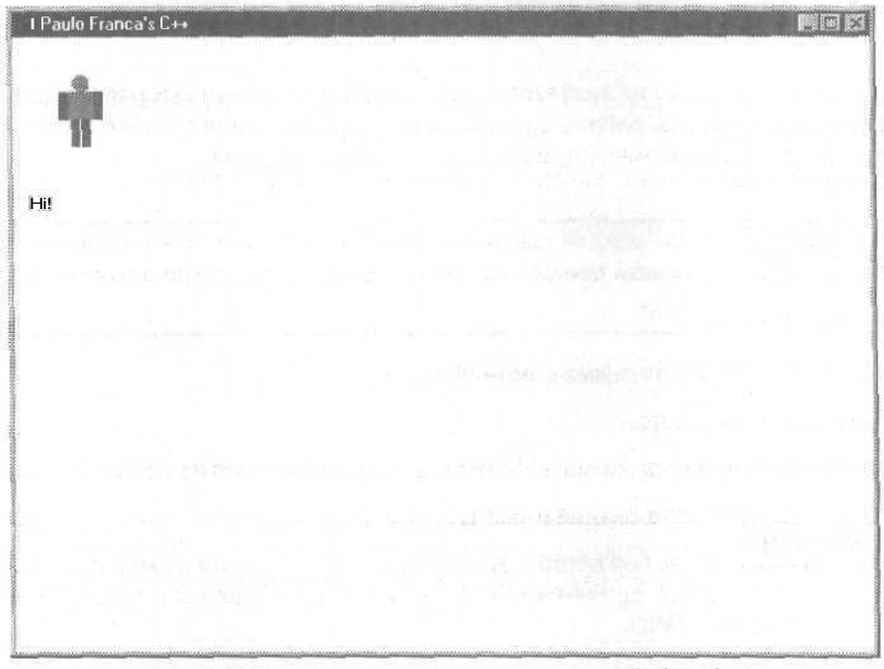


Рис. 0.8. *Результат выполнения программы C_SAL.CPP*

ПРИМЕЧАНИЕ Не приступайте к уроку 1 до тех пор, пока не поймете, как выполнять описанные выше операции.

Файлы проекта

Один из файлов проекта имеет расширение .CPP и определяет, что должна делать программа. Это что-то вроде «персонального модуля» вашего проекта. Чтобы вы-

полнять различные задачи, вы будете заменять одни файлы с расширением .CPP на другие. Эти файлы будут либо примерами, либо созданными вами программами. Остальные файлы обслуживают вашу работу. Файл PAULO.RC называют файлом ресурсов. Он требуется Windows для поддержки диалоговых окон. Файл с расширением .OBJ обеспечивает графическую поддержку ваших программ. Каждый раз, когда в своем проекте вы изменяете файл с исходным кодом программы (файл с расширением .CPP), вы изменяете его «мозг».

Таким образом, чтобы создать программу, необходимо сделать две вещи: написать программу в соответствии с тем, что вы задумали, и поместить эту программу в проект как «персональный модуль».

Различные компиляторы

В следующих разделах объясняется, как использовать различные компиляторы. Выберите раздел, соответствующий вашему компилятору, и прочитайте его, а разделы, относящиеся к другим компиляторам, пропустите.

Программное обеспечение книги тестировалось на следующих компиляторах:

- Turbo C++ 4.5 для Windows.
- Borland C++ 4 для Windows.
- Borland C++ 5 для Windows.
- Microsoft Visual C++ 1.56 для Windows.
- Microsoft Visual C++ 4 для Windows.
- Microsoft Visual C++ 5 для Windows.



ПРИМЕЧАНИЕ Проверьте, не появилась ли на Web-сайте http://www.sybex.com более свежая информация о совместимых компиляторах и соответствующие инструкции.

Borland Turbo C++ 4.5

Если вы используете компилятор Borland Turbo C++ 4.5, то следуйте инструкциям этого раздела.

Запуск компилятора

Для запуска компилятора два раза щелкните на значке Turbo C++ 4.5. Вы увидите окно, показанное на рис. 0.9.

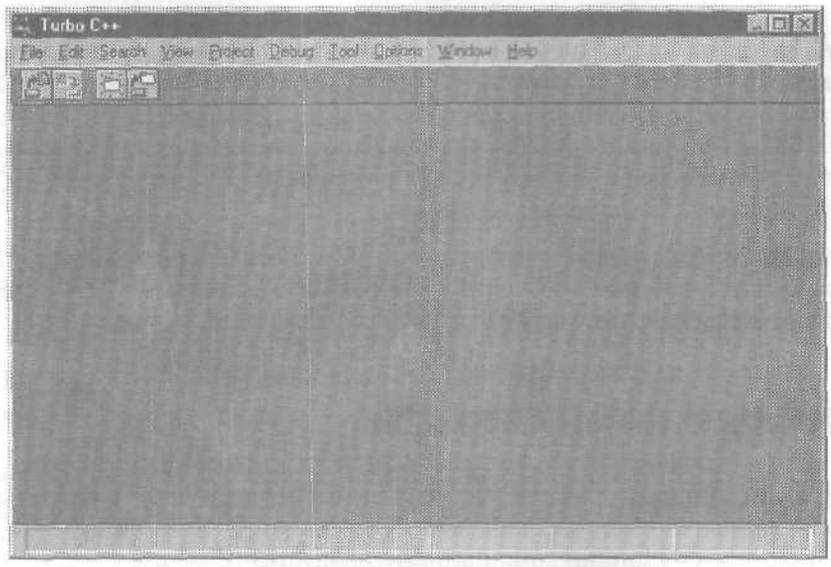


Рис. *0.9. Окно компилятора Turbo C++ 4.5*

Открытие проекта

В верхней части окна вы видите панель меню.

В меню Project выберите команду Open Project (далее мы будем называть это действие «выберите команду Project >Open Project»). Откроется диалоговое окно Open Project File, показанное на рис. 0.10.

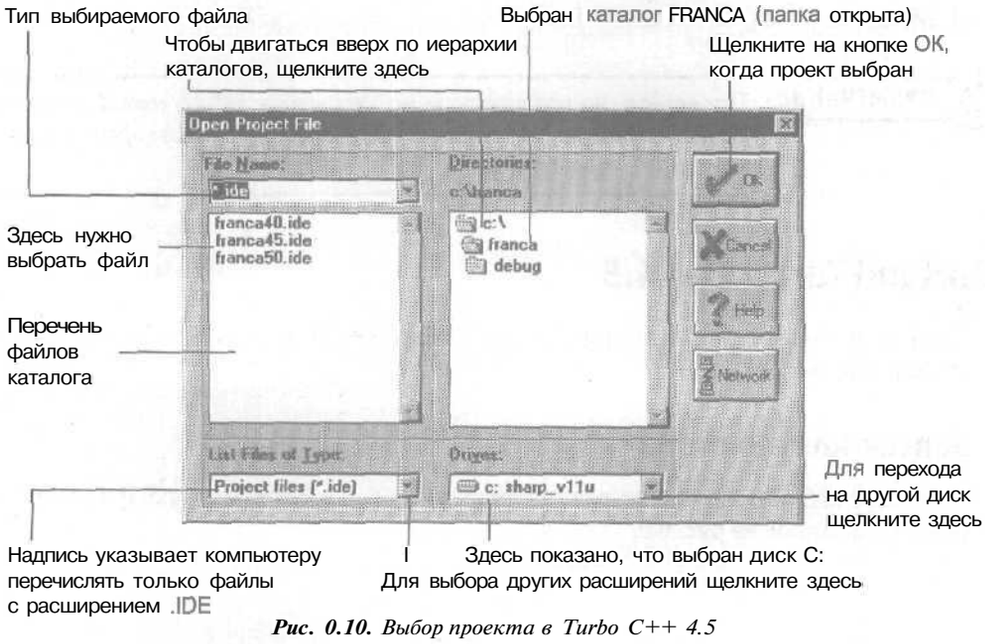


Рис. 0.10. *Выбор проекта в Turbo C++ 4.5*

Убедитесь, что в списке каталогов имеется каталог с программами книги (C:\FRANCA). Если это не так, вы можете либо набрать его имя в поле File Name и щелкнуть на кнопке OK, либо найти его в списке каталогов и щелкнуть на нем мышью.

Проект FRANCA45 подготовлен к работе в среде разработки этого компилятора. Выберите проект, щелкнув на нем два раза (вы также можете щелкнуть на проекте один раз, а затем нажать на кнопку OK).



Когда проект открыт, можете увидеть на экране входящие в него файлы.

Вначале проект должен содержать следующие модули:

- PAULO.RC
- FRANCA45.OBJ
- C_SAL.CPP

Первые два файла (PAULO.RC и FRANCA45.OBJ) должны оставаться в вашем проекте всегда. Последний же (C_SAL.CPP) будет заменяться любой программой, которую вы захотите выполнить. Порядок, в котором перечислены программы, значения не имеет.

Проверьте свои навыки!

Убедитесь, что вы поняли, как нужно работать с компилятором.

1. Запустите ваш компилятор.
2. Выберите проект, с которым вы будете работать.
3. Откройте соответствующий проект в вашем компиляторе.
4. Проверьте, какие файлы содержит ваш проект. Те ли это файлы, которые указаны в тексте?
5. Закройте проект.

Редактирование проекта

В проекте может быть только одна программа с расширением .CPP. Это та самая программа на C++, которую вам понадобится выполнять. Все остальные являются приложениями к ней. На протяжении книги вы будете удалять из проекта одну программу и вставлять на ее место другую. Так вы увидите, как он работает. Исходный проект содержит всего одну программу, которую вы можете просмотреть и которой можете пользоваться.

Удаление программы из проекта

Для удаления файла из проекта щелкните на нем правой кнопкой мыши. Откроется контекстное меню, в котором выберите команду Delete node.

ВНИМАНИЕ Выполняя щелчок на имени программы, которую вы собираетесь удалять, не щелкните случайно на файле FRANCA.EXE, иначе вам придется восстанавливать файл проекта.

Добавление программы в проект

Чтобы добавить в проект программу, щелкните правой кнопкой мыши на имени проекта (FRANCA45.IDE) и затем выберите из контекстного меню команду Add node.

В результате откроется диалоговое окно Add to Project List, показанное на рис. 0.11. Выберите программу, которую вы хотите добавить в проект.

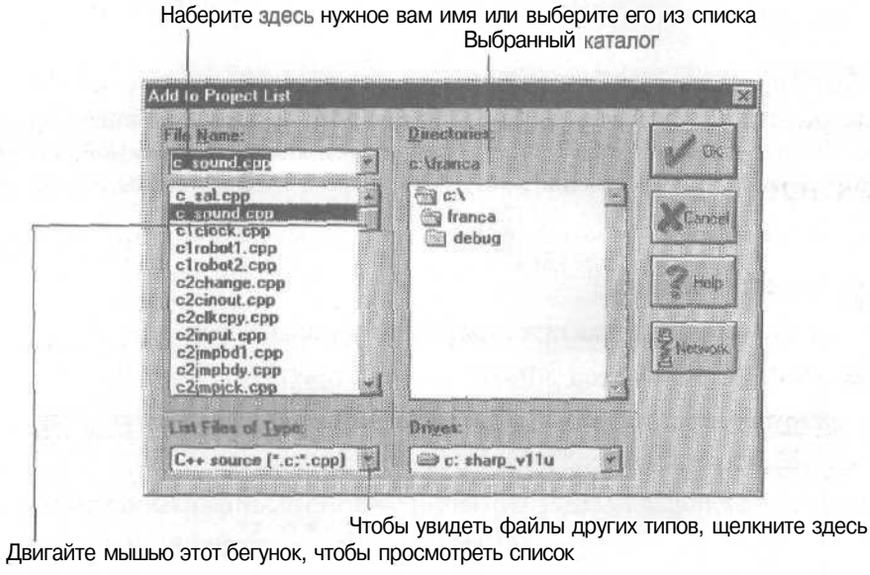


Рис. 0.11. Добавление программы в проект

Выполнение проекта

Чтобы выполнить проект, его надо сначала открыть. Убедитесь, что ваш проект имеет одну (и только одну) программу с расширением .CPP. Для выполнения вашей программы выберите команду Debug ➤ Run.

Если вы выполняете проект с программой C_SAL.CPP, то вы увидите окно, аналогичное показанному на рис. 0.8.

После выполнения программы она остается на экране, пока не будет закрыта. Чтобы закрыть окно программы в Windows 3.1, нужно дважды щелкнуть на значке в

верхнем левом углу окна. Под Windows 95 для закрытия окна программы нужно щелкнуть на квадратике в правом верхнем углу окна.

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Выполните проект и посмотрите на результат. Действительно ли он выглядит так, как показано на рис. 0.8?
3. Закройте окно программы.
4. Посмотрите на список файлов вашего проекта.

   Удалите программу C_SAL.CPP.

   Вставьте программу C_SOUND.CPP.
5. Выполните проект. Если у вас есть звуковое устройство, то вы что-нибудь услышите.
6. Закройте окно программы.
7. Удалите программу C_SOUND.CPP и верните в проект программу C_SAL.CPP.
8. Закройте проект.

Редактирование программ

Часто вам бывает необходимо отредактировать или написать новую программу.

Создание новой программы

Чтобы написать новую программу, выберите команду File ➤ New. Откроется окно, в котором вы сможете набрать исходный код своей программы (рис. 0.12).

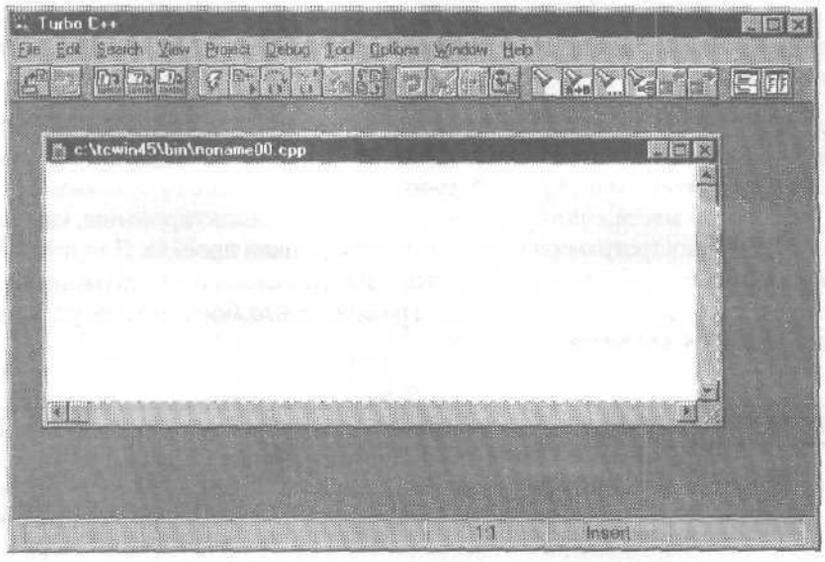


Рис. *0.12.* *Окно новой программы*

Назовем новую программу именем NONAME.CPP и поместим ее в каталог компилятора. Наберите программу и затем выберите команду File > Save As, чтобы открыть диалоговое окно Save File As, показанное на рис. 0.13. Как видно из рисунка, файл действительно называется NONAME.CPP и направляется в каталог компилятора. В поле File Name наберите имя, которое вы хотите дать своему файлу, и поместите его в подходящий каталог, иначе все ваши новые программы останутся в каталоге компилятора.

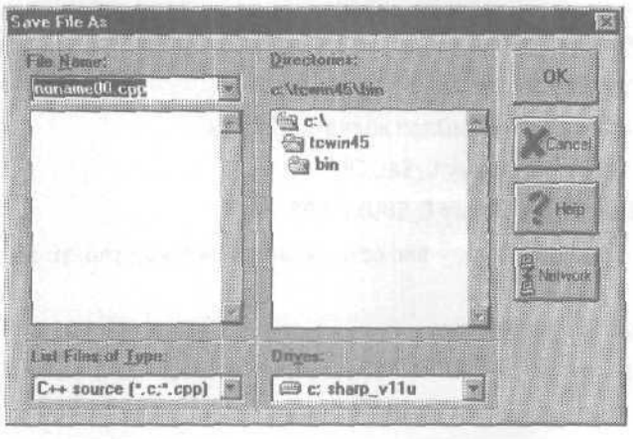


Рис. 0.13. *Выбор имени и каталога программы*

Редактирование существующей программы

Чтобы просмотреть и отредактировать программу, не входящую в открытый в данный момент проект, выберите команду File ➤ Open. Откроется диалоговое окно File Open. Если вы увидели там нужный вам файл, щелкните два раза на его имени. В противном случае убедитесь, что вы правильно выбрали каталог. Обычно компьютер отображает файлы с расширением .CPP (то есть файлы с исходными кодами программ). Если же вам нужны файлы с другими расширениями, укажите это в данном окне.

Чтобы просматривать и редактировать программу, относящуюся к открытому проекту, дважды щелкните на ее имени (например, на файле C_SAL.CPP). Откроется окно, соответствующее выбранной программе. Когда окно программы откроется, щелкните на том месте, с которого вы хотите начать редактирование, как показано на рис. 0.14. Окно программы появляется поверх окна проекта. Для перемещения любого окна нажмите левую кнопку мыши на его заголовке и, удерживая ее нажатой, переместите окно. Вы также можете разместить любое окно поверх остальных, щелкнув в любом его месте.

ВНИМАНИЕ Модифицируя программу, вы утрачиваете ее оригинал. Если вы не хотите этого, то выберите команду File ➤ Save As и сохраните программу под другим именем.

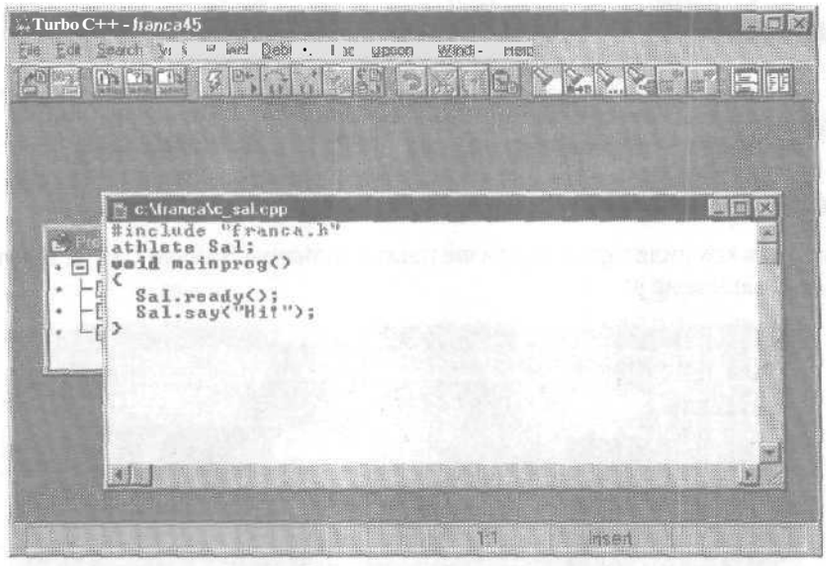


Рис. *0.14. Редактирование программы*

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Проверьте все файлы в вашем проекте и убедитесь, что файл C_SAL.CPP является единственным файлом с расширением .CPP. Если это не так, то удалите все другие файлы с расширением .CPP и вместо них вставьте файл C_SAL.CPP.
3. Откройте окно программы C_SAL.CPP.
4. На предпоследней строке написано: Sal say («Hi!») (Сэл сказал («Хай!»)); замените Hi! на Hello! (установите ваш курсор справа перед символом H и нажмите левую кнопку мыши. Затем наберите Hello и нажмите на клавишу Delete несколько раз, чтобы удалить первоначальное Hi).
5. Сохраните новую версию программы под именем C_SALNEW.CPP. Убедитесь, что вы сохранили эту программу в том же каталоге, в котором находятся остальные программы (каталог C:\FRANCA). Если вы выполните теперь ваш проект, как вы думаете, что скажет Сэл? «Hi» или «Hello»? Сэл будет по-прежнему говорить «Hi», поскольку программа C_SALNEW.CPP еще не включена в проект. Если вы мне не верите, запустите программу.
6. Удалите файл C_SAL.CPP из проекта и поместите туда файл C_SALNEW.CPP.
7. Выполните проект. Говорит ли теперь Сэл «Hello»?
8. Удалите файл C_SALNEW.CPP из проекта и верните файл C_SAL.CPP обратно.

Borland C++ 4

Если вы используете компилятор Borland C++ 4, то следуйте инструкциям этого раздела.

Запуск компилятора

Для запуска компилятора два раза щелкните на значке Borland C++ 4. Вы увидите окно, показанное на рис. 0.15.

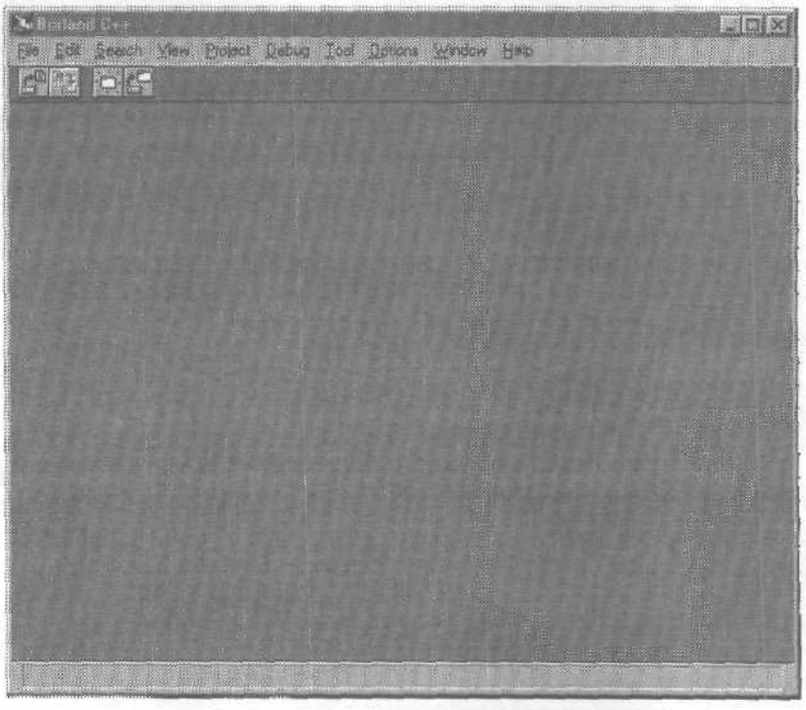


Рис. 0,15. *Окно компилятора Borland C++ 4*

Открытие проекта

В верхней части окна вы видите панель меню.

Выберите команду Project ➤ Open Project. Откроется диалоговое окно Open Project File, показанное на рис. 0.16.

Убедитесь, что в списке каталогов имеется каталог с программами книги (C:\FRANCA). Если это не так, то вы можете либо набрать его имя в поле File Name и щелкнуть на кнопке OK, либо найти его в списке каталогов и щелкнуть на нем мышью.

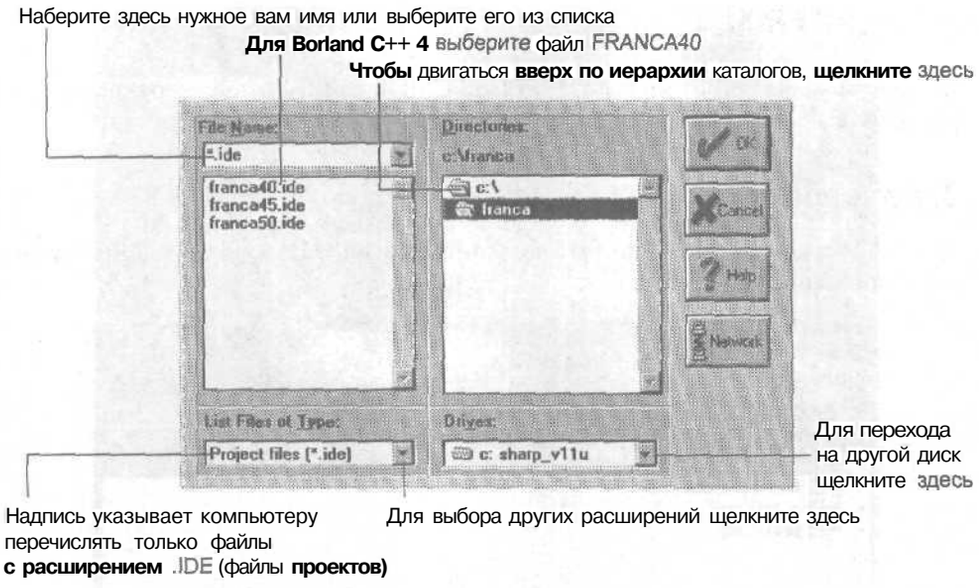


Рис. 0.16. *Выбор проекта в компиляторе Borland C++ 4*

Проект FRANCA40 подготовлен к работе в среде разработки этого компилятора. Выберите проект, щелкнув на нем два раза (вы также можете щелкнуть на проекте один раз, а затем щелкнуть на кнопке OK).

Когда проект будет открыт, вы сможете увидеть на экране содержащуюся в нем информацию, как показано на рис. 0.17.

Вначале проект будет содержать следующие модули:

- PAULO.RC
- FRANCA40.OBJ
- C_SAL.CPP

Первые два файла (PAULO.RC и FRANCA40.OBJ) должны оставаться в вашем проекте всегда. Последний же (C_SAL.CPP) будет заменяться любой программой, которую вы захотите выполнить. Порядок, в котором перечисляются программы, значения не имеет.

Проверьте свои навыки!

Убедитесь, что вы поняли, как нужно работать с компилятором.

1. Запустите ваш компилятор.
2. Выберите проект, с которым вы будете работать.
3. Откройте соответствующий проект в вашем компиляторе.

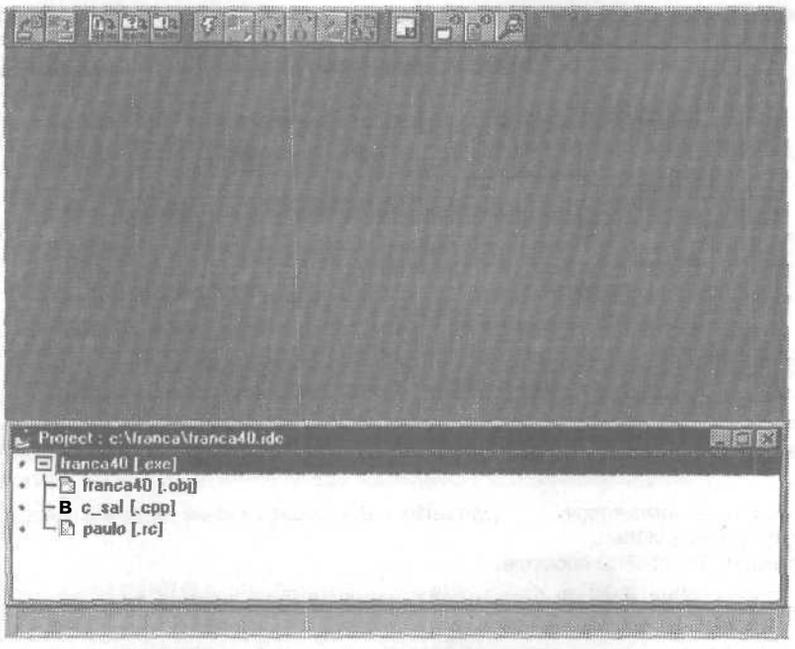


Рис. *0.17.* *Окно проекта FRANCA40для Borland C++ 4*

4. Проверьте, какие файлы содержит ваш проект. Те ли это файлы, которые указаны в тексте?
5. Закройте проект.

Редактирование проекта

В проекте может быть только одна программа с расширением .CPP. Это та самая программа на C++, которую вам понадобится выполнять. Все остальные являются приложениями к ней. На протяжении книги вы будете удалять из проекта одну программу и вставлять на ее место другую. Так вы увидите, как он работает. Исходный проект содержит всего одну программу, которую вы можете просмотреть и которой можете воспользоваться.

Удаление программы из проекта

Для удаления файла из проекта щелкните на нем правой кнопкой. Откроется контекстное меню, в котором выберите команду Delete node.

ВНИМАНИЕ Выполняя щелчок на имени программы, которую вы собираетесь удалять, не щелкните случайно на файле FRANCA40.EXE, иначе вам придется восстанавливать файл проекта.

Добавление программы в проект

Чтобы добавить в проект программу, щелкните правой кнопкой мыши на имени проекта (FRANCA40.EXE) и затем выберите из контекстного меню команду Add node.

Теперь выберите в списке файл, который вы хотите добавить в проект.

Выполнение проекта

Чтобы выполнить проект, его надо сначала открыть. Убедитесь, что ваш проект имеет одну (и только одну) программу с расширением .CPP. Для выполнения вашей программы выберите команду Debug➤Run.

Если вы выполняете проект с программой C_SAL.CPP, вы увидите окно, аналогичное показанному на рис. 0.8.

После выполнения программы она остается на экране, пока не будет закрыта. Чтобы закрыть окно программы в Windows 3.1, нужно дважды щелкнуть на значке в верхнем левом углу окна. Под Windows 95 для закрытия окна программы нужно щелкнуть на квадратике в правом верхнем углу окна.

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP:

1. Если проект не открыт, откройте его.
2. Выполните проект и посмотрите на результат. Действительно ли он выглядит так, как показано на рис. 0.8?
3. Закройте окно программы,
4. Посмотрите на список файлов вашего проекта.

   Удалите программу C_SAL.CPP.

   Вставьте программу C_SOUND.CPP.
5. Выполните проект. Если у вас есть звуковое устройство, то вы что-нибудь услышите.
6. Закройте окно программы.
7. Удалите программу C_SOUND.CPP и верните в проект программу C_SAL.CPP.
8. Закройте проект.

Редактирование программ

Часто вам бывает необходимо отредактировать существующую или написать новую программу.

Создание новой программы

Чтобы написать новую программу, выберите команду File > New. Откроется окно, в котором вы сможете набрать исходный код своей программы (рис. 0.18). Введите программу, а затем выберите команду File > Save As, чтобы записать ее на диск.

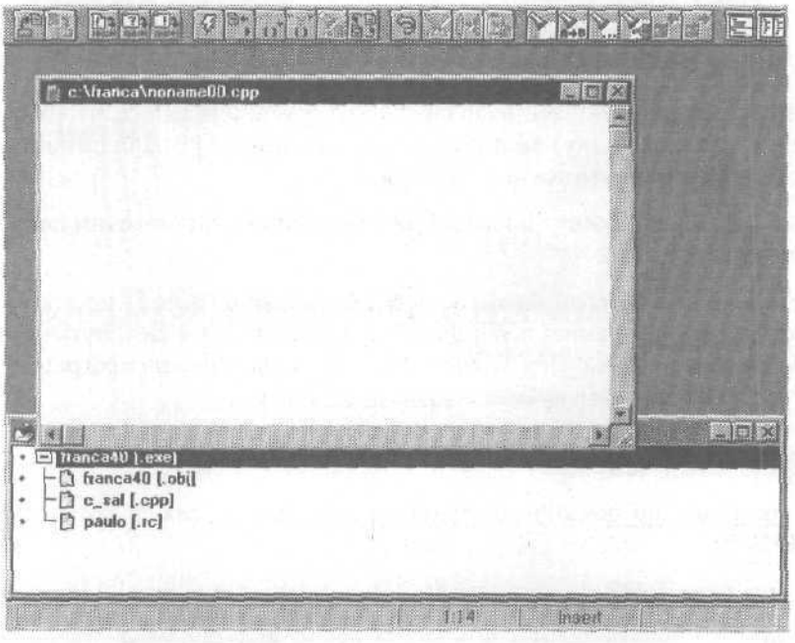


Рис. 0.18. *Окно новой программы (NONAME00.CPP)*

Редактирование существующей программы

Чтобы просмотреть и отредактировать программу, не входящую в открытый в данный момент проект, выберите команду File ➤ Open. Откроется диалоговое окно File Open. Если вы увидели там нужный вам файл, щелкните два раза на его имени. В противном случае убедитесь, что вы правильно выбрали каталог. Обычно компьютер отображает файлы с расширением .CPP (исходные файлы). Если же вам нужны файлы с другими расширениями, укажите это в данном окне.

Чтобы просматривать и редактировать программу, относящуюся к открытому проекту, дважды щелкните на ее имени. Откроется окно, соответствующее выбранной программе. Когда окно программы откроется, щелкните на том месте, с которого вы хотите начать редактирование, как показано на рис. 0.19.

Окно программы появляется поверх окна проекта. Для перемещения любого окна нажмите левую кнопку мыши на его заголовке и, удерживая ее нажатой, переместите окно. Вы также можете разместить любое окно поверх остальных, щелкнув в любом его месте.

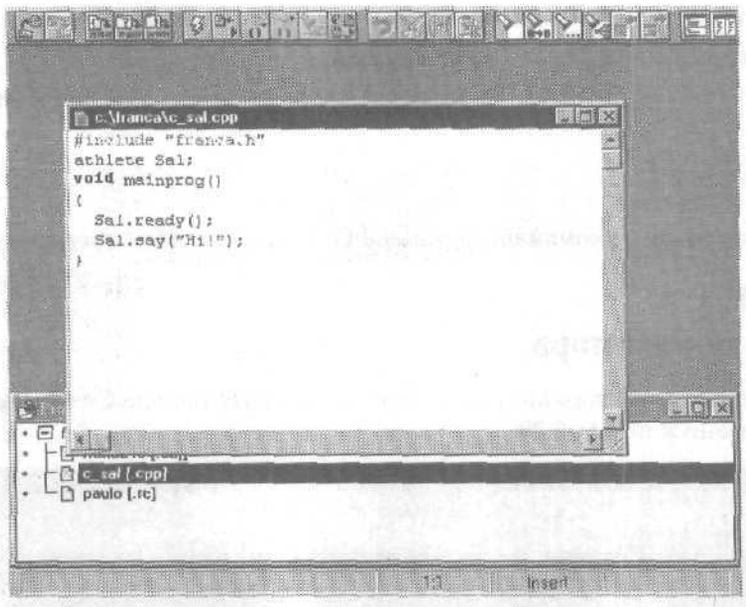


Рис. *0.19. Редактирование программы*

ВНИМАНИЕ Модифицируя программу, вы утрачиваете ее оригинал. Если вы не хотите этого, то выберите команду File ➤ Save As и сохраните программу под другим именем.

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Проверьте все файлы в вашем проекте и убедитесь, что файл C_SAL.CPP является единственным файлом с расширением .CPP. Если это не так, то удалите все другие файлы с расширением .CPP и вместо них вставьте файл C_SAL.CPP.
3. Откройте окно программы C_SAL.CPP.
4. На предпоследней строке написано: Sal say («Hi!») (Сэл сказал («Хай!»)); замените Hi! на Hello! (установите ваш курсор справа перед символом H и нажмите левую кнопку мыши. Затем наберите Hello и нажмите на клавишу Delete несколько раз, чтобы удалить первоначальное Hi).
5. Сохраните новую версию программы под именем C_SALNEW.CPP. Убедитесь, что вы сохранили эту программу в том же каталоге, в котором находятся остальные программы (каталог C:\FRANCA). Если вы выполните теперь ваш проект, как вы думаете, что скажет Сэл? «Hi» или «Hello»? Сэл будет по-прежнему говорить «Hi», поскольку программа C_SALNEW.CPP еще не включена в проект. Если вы мне не верите, запустите программу.

6. Удалите файл C_SAL.CPP из проекта и поместите туда файл C_SALNEW.CPP.
7. Выполните проект. Говорит ли теперь Сэл «Hello»?
8. Удалите файл C_SALNEW.CPP из проекта и верните файл C_SAL.CPP обратно.

Borland C++ 5

Если вы используете компилятор Borland C++ 5, то следуйте инструкциям этого раздела.

Запуск компилятора

Для запуска компилятора два раза щелкните на значке Borland C++ 4. Вы увидите окно, показанное на рис. 0.20.

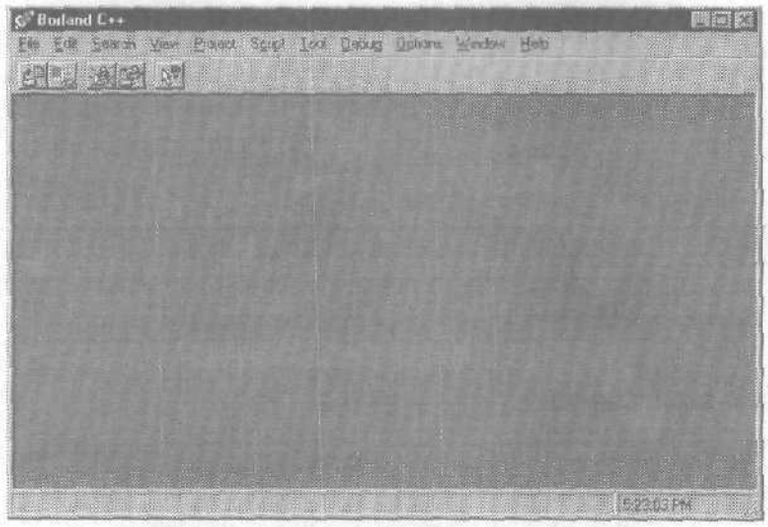


Рис. 0.20. *Окно компилятора Borland C++ 5*

Открытие проекта

В верхней части окна вы видите панель меню.

Выберите команду Project ➤ Open Project. Откроется диалоговое окно Open Project File, показанное на рис. 0.21.

Убедитесь, что в списке каталогов имеется каталог с программами книги (C:\FRANCA). Если это не так, то вы можете либо набрать его имя в поле File Name и щелкнуть на кнопке OK, либо найти его в списке каталогов и щелкнуть на нем мышью.

Проект FRANCA50 подготовлен к работе в среде разработки компилятора Borland C++ 5. Выберите проект, щелкнув на нем два раза (вы также можете щелкнуть на проекте один раз, а затем щелкнуть на кнопке OK), чтобы открыть окно, показанное на рис. 0.22.

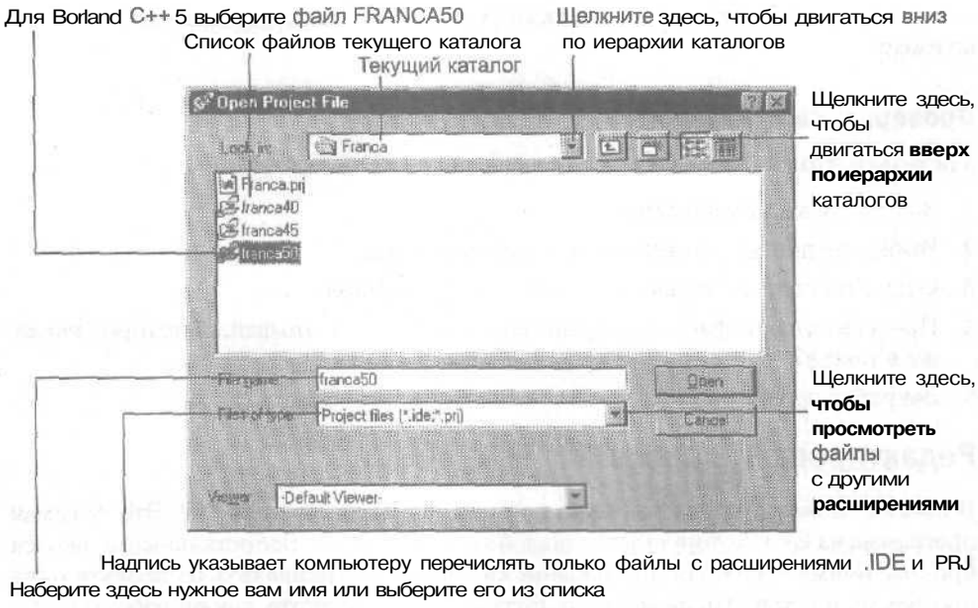


Рис. 0.21. *Выбор проекта в компиляторе Borland C++ 5*

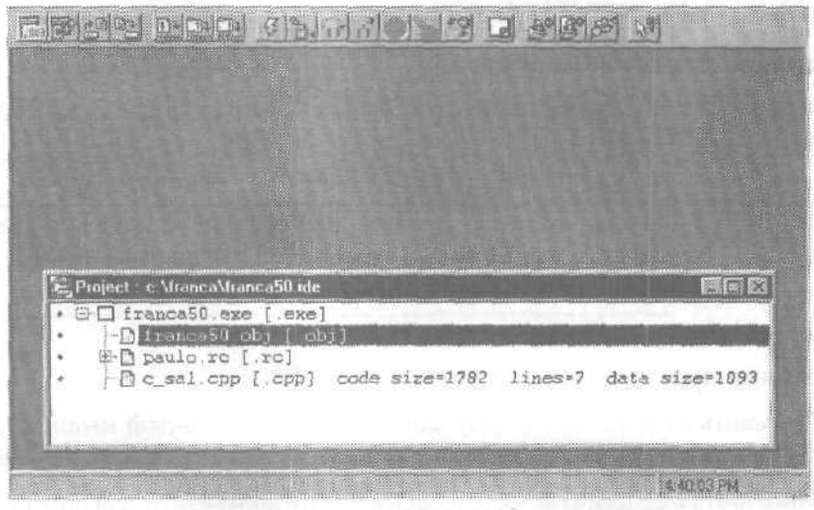


Рис. 0.22. *Окно проекта FRANCA50 для Borland C++ 5*

Вначале проект будет содержать следующие модули:

- PAULO.RC
- FRANCA50.OBJ
- C_SAL.CPP

Первые два файла (PAULO.RC и FRANCA50.OBJ) должны оставаться в вашем проекте всегда. Последний же (C_SAL.CPP) будет заменяться любой программой, которую

вы захотите выполнить. Порядок, в котором перечисляются программы, значения не имеет.

Проверьте свои навыки!

Убедитесь, что вы поняли, как нужно работать с компилятором.

1. Запустите ваш компилятор.
2. Выберите проект, с которым вы будете работать.
3. Откройте соответствующий проект в вашем компиляторе.
4. Проверьте, какие файлы содержит ваш проект. Те ли это файлы, которые указаны в тексте?
5. Закройте проект.

Редактирование проекта

В проекте может быть только одна программа с расширением .CPP. Это та самая программа на C++, которую вам понадобится выполнять. Все остальные являются приложениями к ней. На протяжении книги вы будете удалять из проекта одну программу и вставлять на ее место другую. Так вы увидите, как он работает. Исходный проект содержит всего одну программу, которую вы можете просмотреть и которой можете воспользоваться.

Удаление программы из проекта

Для удаления файла из проекта щелкните на нем правой кнопкой. Откроется контекстное меню, в котором выберите команду Delete node.



ВНИМАНИЕ Выполняя щелчок на имени программы, которую вы собираетесь удалять, не щелкните случайно на файле FRANCA50.EXE, иначе вам придется восстанавливать файл проекта.

Добавление программы в проект

Чтобы добавить в проект программу, щелкните правой кнопкой мыши на имени проекта (FRANCA50.EXE) и затем выберите из контекстного меню команду Add node.

Теперь выберите в списке файл, который вы хотите добавить в свой проект.

В результате откроется диалоговое окно Add to Project List, показанное на рис. 0.23. Выберите в нем нужный файл.

Выполнение проекта

Чтобы выполнить проект, его надо сначала открыть. Убедитесь, что ваш проект имеет одну (и только одну) программу с расширением .CPP. Для выполнения вашей программы выберите команду Debug ► Run.

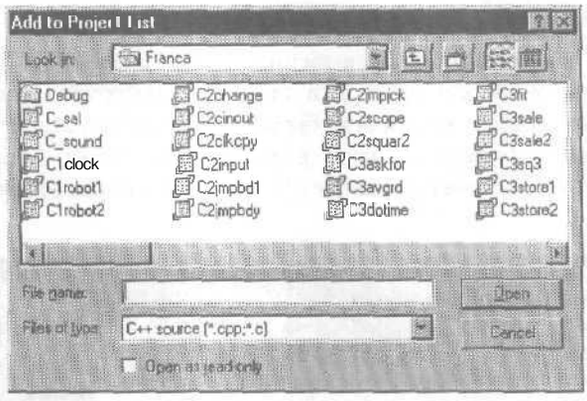


Рис. *0,23.* *Выбор программ для проекта*

Если вы выполняете проект с программой C_SAL.CPP, то увидите окно, аналогичное показанному на рис. 0.8.

После выполнения программы она остается на экране, пока не будет закрыта. Чтобы закрыть окно программы в Windows 3.1, нужно дважды щелкнуть на значке в верхнем левом углу окна. Под Windows 95 для закрытия окна программы нужно щелкнуть на квадратике в правом верхнем углу окна.

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Выполните проект и посмотрите на результат. Действительно ли он выглядит так, как показано на рис. 0.8?
3. Закройте окно программы.
4. Посмотрите на список файлов вашего проекта.

   Удалите программу C_SAL.CPP.

   Вставьте программу C_SOUND.CPP.
5. Выполните проект. Если у вас есть звуковое устройство, то вы что-нибудь услышите.
6. Закройте окно программы.
7. Удалите программу C_SOUND.CPP иверните в проект программу C_SAL.CPP.
8. Закройте проект.

Редактирование программ

Часто вам бывает необходимо отредактировать существующую или написать новую программу.

Создание новой программы

Чтобы написать новую программу, выберите команду File ➤ New. Откроется окно, Б котором вы сможете набрать исходный код своей программы (рис. 0.24). Введите программу, а затем выберите команду File ➤ Save As, чтобы записать ее на диск. При этом не забудьте поместить файл в выбранный вами каталог (например, C:\FRANCA) и дать ему соответствующее имя, набрав его в поле File name, как показано на рис. 0,25.

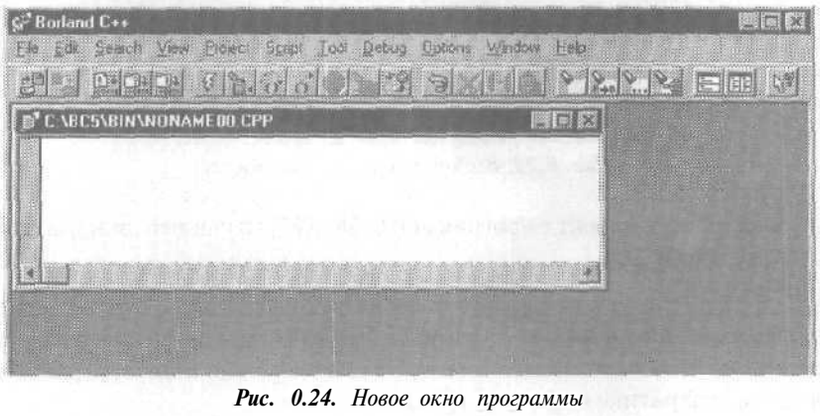


Рис. 0.24. *Новое окно программы*

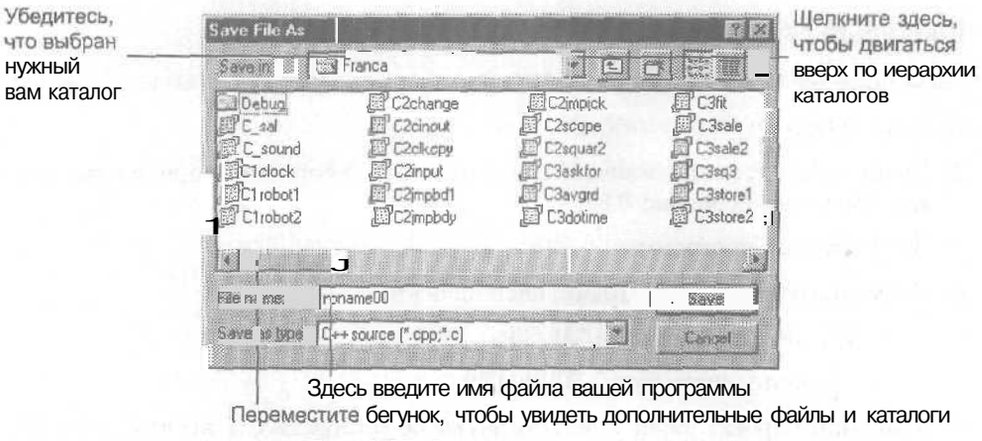


Рис. 0.25. *Сохранение программы*

Редактирование существующей программы

Чтобы просмотреть и отредактировать программу, не входящую в открытый в данный момент проект, выберите команду File ➤ Open. Откроется диалоговое окно File Open. Если вы увидели там нужный вам файл, щелкните два раза на его имени. В противном случае убедитесь, что вы правильно выбрали каталог. Обычно компьютер отображает файлы с расширением .CPP (исходные файлы). Если же вам нужны файлы с другими расширениями, укажите это в данном окне.

Чтобы просматривать и редактировать программу, относящуюся к открытому проекту, дважды щелкните на ее имени. Откроется окно, соответствующее выбранной программе. Когда окно программы откроется, щелкните на том месте, с которого вы хотите начать редактирование, как показано на рис. 0.26.

Окно программы появляется поверх окна проекта. Для перемещения любого окна нажмите левую кнопку мыши на его заголовке и, удерживая ее нажатой, переместите окно. Вы также можете разместить любое окно поверх остальных, щелкнув в любом его месте.

ВНИМАНИЕ Модифицируя программу, вы *утрачиваете* ее оригинал. Если вы не хотите этого, то выберите команду File➤ Save As и сохраните программу под другим именем.

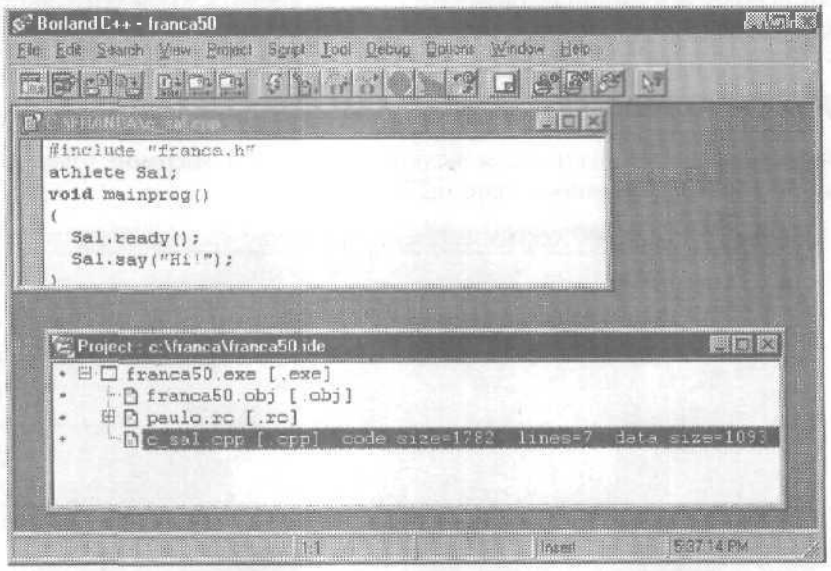


Рис. 0.26. Открытие фата C_SAL.CPP в Borland C++ 5

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Проверьте все файлы в вашем проекте и убедитесь, что файл C_SAL.CPP является единственным файлом с расширением .CPP. Если это не так, то удалите все другие файлы с расширением .CPP и вместо них вставьте файл C_SAL.CPP.
3. Откройте окно программы C_SAL.CPP.
4. На предпоследней строке написано: Sal say («Hi!») (Сэл сказал («Хай!»)); замените Hi! на Hello! (установите ваш курсор справа перед символом H и нажмите левую кнопку мыши. Затем наберите Hello и нажмите на клавишу Delete несколько раз, чтобы удалить первоначальное Hi).

5. Сохраните новую версию программы под именем C_SALNEW.CPP. Убедитесь, что вы сохранили эту программу в том же каталоге, в котором находятся остальные программы (каталог C:\FRANCA). Если вы выполните теперь ваш проект, как вы думаете, что скажет Сэл? «Hi» или «Hello»? Сэл будет по-прежнему говорить «Hi», поскольку программа C_SALNEW.CPP еще не включена в проект. Если вы мне не верите, запустите программу.
6. Удалите файл C_SAL.CPP из проекта и поместите туда файл C_SALNEW.CPP.
7. Выполните проект. Говорит ли теперь Сэл «Hello»?
8. Удалите файл C_SALNEW.CPP из проекта и верните файл C_SAL.CPP обратно.

Microsoft Visual C++ 1.5

Если вы используете компилятор Microsoft Visual C++ 1.5, то следуйте инструкциям этого раздела.

Запуск компилятора

Для запуска компилятора дважды щелкните на значке Microsoft Visual C++ 1.5. Вы увидите окно, показанное на рис. 0.27.

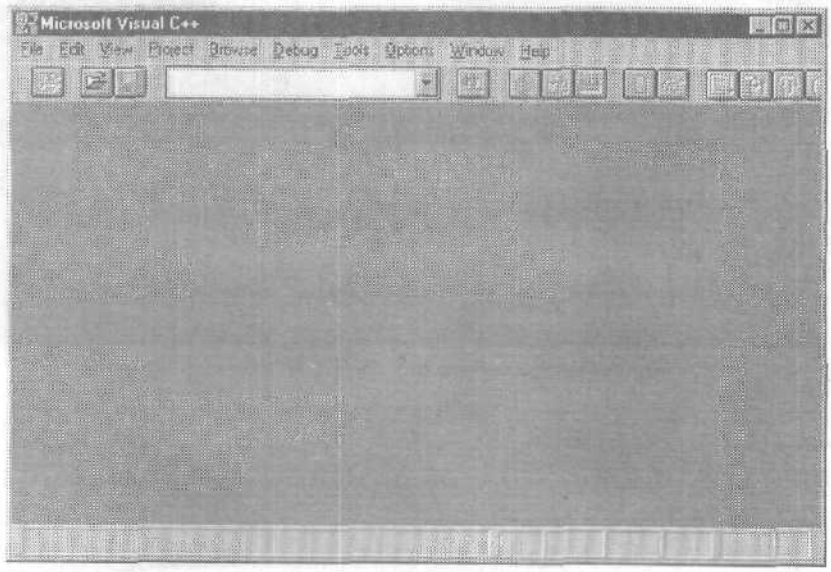


Рис. 0,27. *Окно компилятора Microsoft Visual C++ 1.5*

Открытие проекта

В верхней части окна вы видите **панель меню.**

Выберите команду Project ➤ Open Project, Откроется диалоговое окно Open Project, показанное на рис. 0.28.

Убедитесь, что в списке каталогов имеется каталог с программами книги (C:\FRANCA). Если это не так, то вы можете набрать его имя в поле File name или выбрать из списка каталогов.

Проект FRANCAM1 подготовлен к работе в среде разработки компилятора Visual C++ 1.5. Выберите проект, щелкнув на нем два раза (вы также можете щелкнуть на проекте один раз, а затем щелкнуть на кнопке OK).

Когда проект открыт, для просмотра входящих в проект файлов выберите команду Project ➤ Edit. Откроется диалоговое окно Edit, показанное на рис. 0.29).

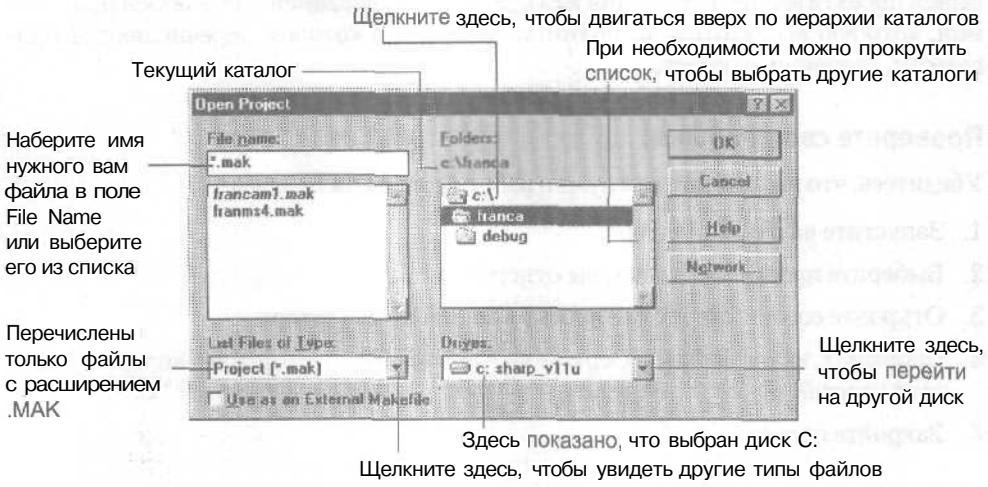


Рис. 0.28. *Диалоговое окно Open Project компилятора Visual C++ 1.5*

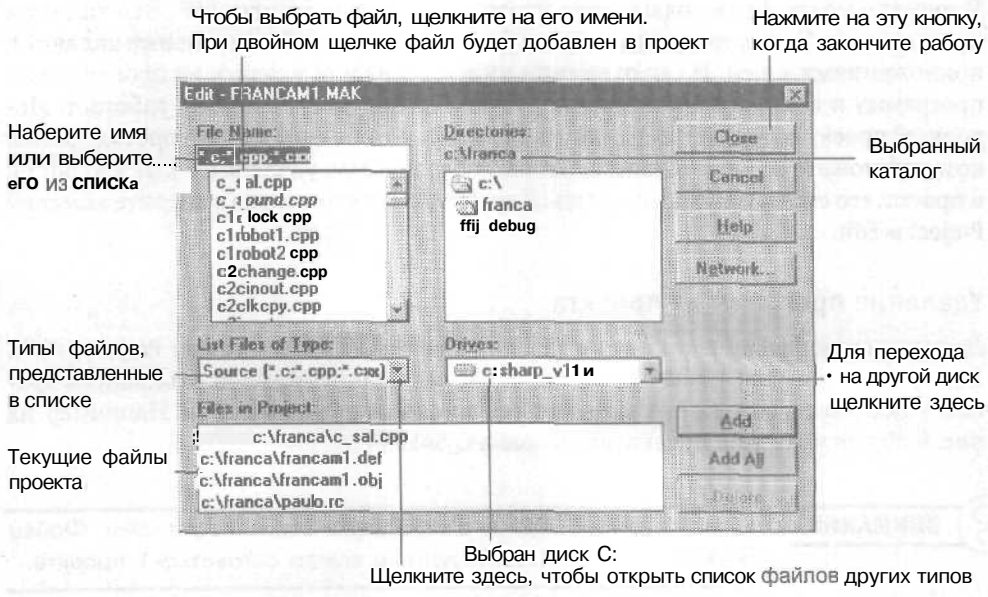


Рис. 0.29. *Окно редактирования проекта*

Вначале проект будет содержать следующие модули:

- PAULO.RC
- FRANCAM1.OBJ
- FRANCA.DEF
- C_SAL.CPP

Первые три файла (PAULO.RC, FRANCAM1.OBJ и FRANCAM1.DEF) должны оставаться в вашем проекте всегда. Последний же (C_SAL.CPP) будет заменяться любой программой, которую вы захотите выполнить. Порядок, в котором перечисляются программы, значения не имеет.

Проверьте свои навыки!

Убедитесь, что вы поняли, как нужно работать с компилятором.

1. Запустите ваш компилятор.
2. Выберите проект, с которым вы будете работать.
3. Откройте соответствующий проект в вашем компиляторе.
4. Проверьте, какие файлы содержит ваш проект. Те ли это файлы, которые указаны в тексте?
5. Закройте проект.

Редактирование проекта

В проекте может быть только одна программа с расширением .CPP. Это та самая программа на C++, которую вам понадобится выполнять. Все остальные являются приложениями к ней. На протяжении книги вы будете удалять из проекта одну программу и вставлять на ее место другую. Так вы увидите, как он работает. Исходный проект содержит всего одну программу, которую вы можете просмотреть и которой можете воспользоваться. Чтобы добавлять или удалять файлы, входящие в проект, его сначала нужно открыть. Для этого в главном меню выберите команду Project ➤ Edit.

Удаление программ из проекта

Для удаления файла из проекта в главном меню выберите команду Project ➤ Edit. В открывшемся окне нижнее левое поле содержит список файлов, входящих в проект. Удаление любого из них выполняется двойным щелчком мыши. Например, на рис. 0.30 для удаления предназначен файл C_SAL.CPP.

ВНИМАНИЕ Удаляйте только файлы с исходными кодами программ. Файлы FRANCAM1.OBJ и PAULO.RC должны всегда оставаться в проекте.

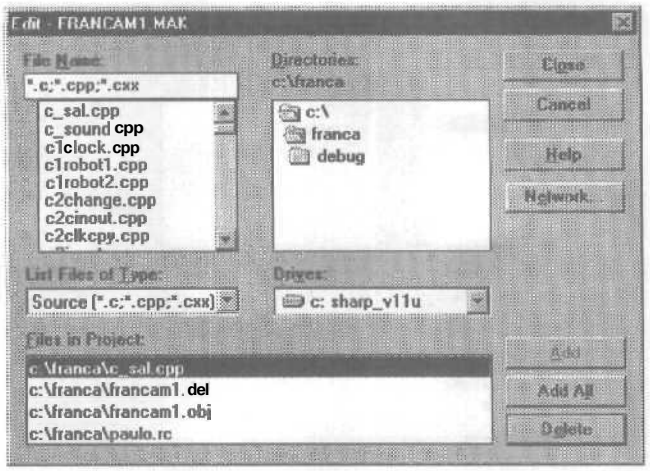


Рис. 0.30. Удаление файла C_SAL.CPP

После удаления файла C_SAL.CPP диалоговое окно Edit примет вид, показанный на рис. 0.31.

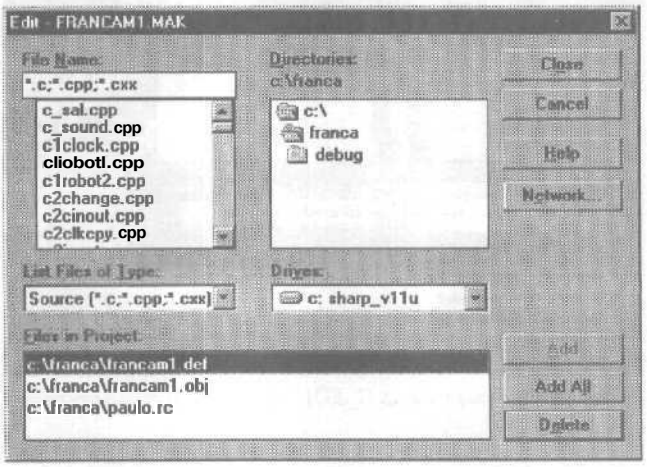


Рис. 0.31. Проект после удаления файла C_SAL.CPP

Добавление программы в проект

Чтобы добавить в проект программу, проект нужно сначала открыть. Для этого в главном меню выберите команду Project ➤ Edit. Вы увидите окно Edit, показанное на рис. 0.31. Левое верхнее поле содержит список файлов в текущем каталоге. Выберите тот, который вы хотите включить в проект, и дважды щелкните на нем мышью. Например, если вам нужен файл C_SOUND.CPP, то, выбрав его (как показано на рис. 0.32), нажмите кнопку Add. Вы увидите новое содержимое проекта, как показано на рис. 0.33.

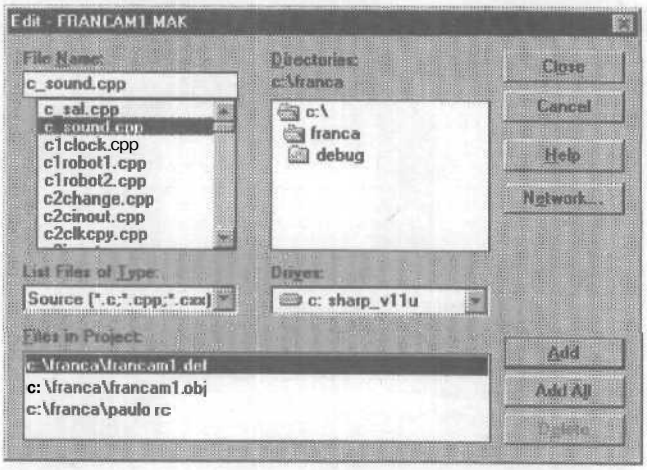


Рис. 0.32. *Выбор файла C_SOUND.CPP для включения в проект*

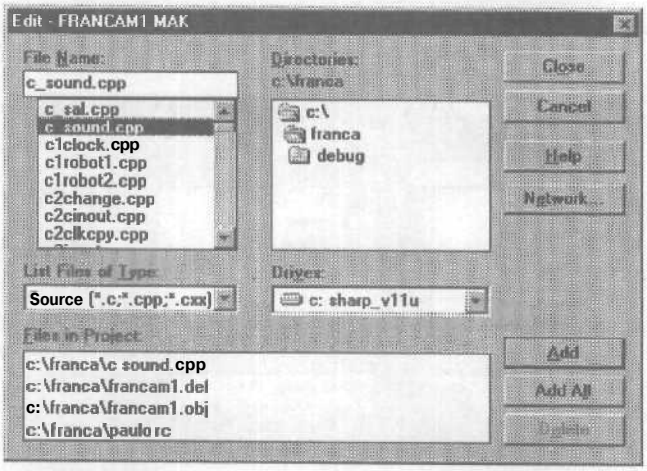


Рис. *0.33. Теперь файл C_SOUND,CPP включен в проект*

Выполнение проекта

Чтобы выполнить проект, его надо сначала открыть. Убедитесь, что в вашем проекте имеется одна (и только одна) программа с расширением .CPP.

Visual C++ сначала создает проект (собирает вместе все файлы), а затем предоставляет вам возможность запустить его. Чтобы создать проект, выберите в главном меню команду Project ➤Build, а когда проект будет создан - команду Debug ➤Go.

Если вы выполняете проект с программой C_SAL.CPP, то увидите экран, аналогичный тому, который показан на рис. 0.8. При этом Сэл поприветствует вас. Попробуйте!

После выполнения программы она остается на экране, пока не будет закрыта. Чтобы закрыть окно программы в Windows 3.1, нужно дважды щелкнуть на значке в верхнем левом углу окна. Под Windows 95 для закрытия окна программы нужно щелкнуть на квадратике в правом верхнем углу окна.



ВНИМАНИЕ Для выполнения проекта достаточно сразу выбрать команду Debug ➤ Go. В этом случае Visual C++ спросит вас, будете ли вы создавать проект (а вы, конечно, будете), а затем после создания проекта остановится, и вам опять придется выбирать Debug ➤ Go.

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Выполните проект и посмотрите на результат. Действительно ли он выглядит так, как показано на рис. 0.8?
3. Закройте окно программы.
4. Посмотрите на список файлов вашего проекта.

   Удалите программу C_SAL.CPP.

   Вставьте программу C_SOUND.CPP.
5. Выполните проект. Если у вас есть звуковое устройство, то вы что-нибудь услышите.
6. Закройте окно программы.
7. Удалите программу C_SOUND.CPP и верните в проект программу C_SAL.CPP.
8. Закройте проект.

Редактирование программ

Часто бывает необходимо отредактировать существующую или написать новую программу.

Создание новой программы

Чтобы написать новую программу, выберите команду File ➤ New. Откроется окно, в котором вы сможете набрать исходный код своей программы (рис. 0.34). Введите программу, а затем выберите команду File ➤ Save As, чтобы записать ее на диск. При этом не забудьте поместить файл в выбранный вами каталог и дать ему соответствующее имя, набрав его в поле File Name, как показано на рис. 0.35.

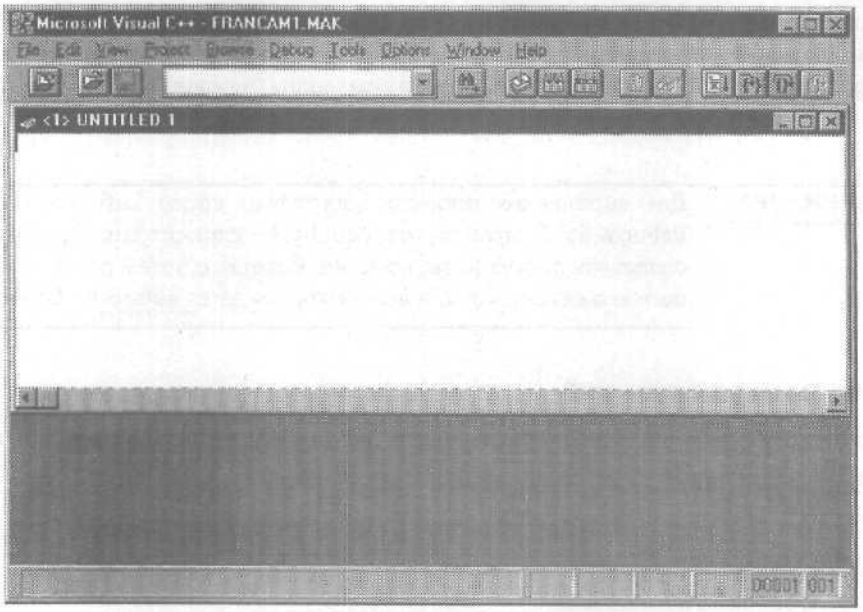


Рис. 0.34. Окно новой программы

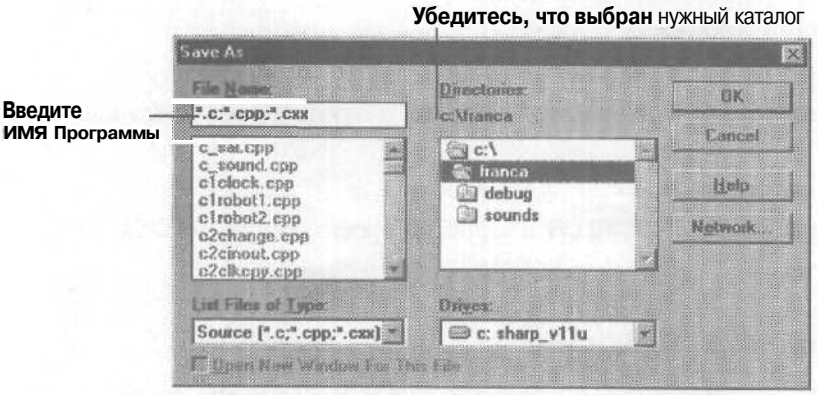


Рис. *0.35. Сохранение программы*

Редактирование существующей программы

Чтобы просмотреть и отредактировать программу, не входящую в открытый в данный момент проект, выберите команду File ➤ Open. Откроется диалоговое окно Open File. Если вы увидели там нужный вам файл, щелкните два раза на его имени. В противном случае убедитесь, что вы правильно выбрали каталог. Обычно компьютер отображает файлы с расширением .CPP (исходные файлы). Если же вам нужны файлы с другими расширениями, укажите это в данном окне.

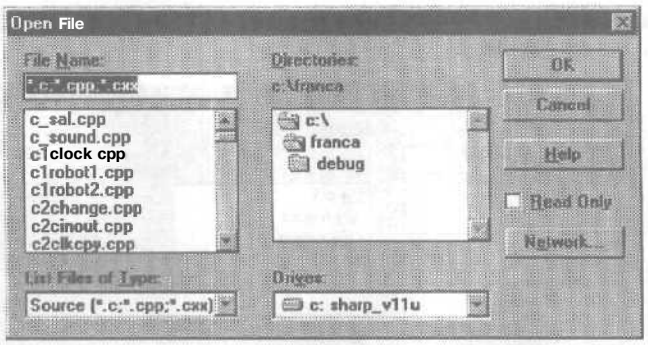
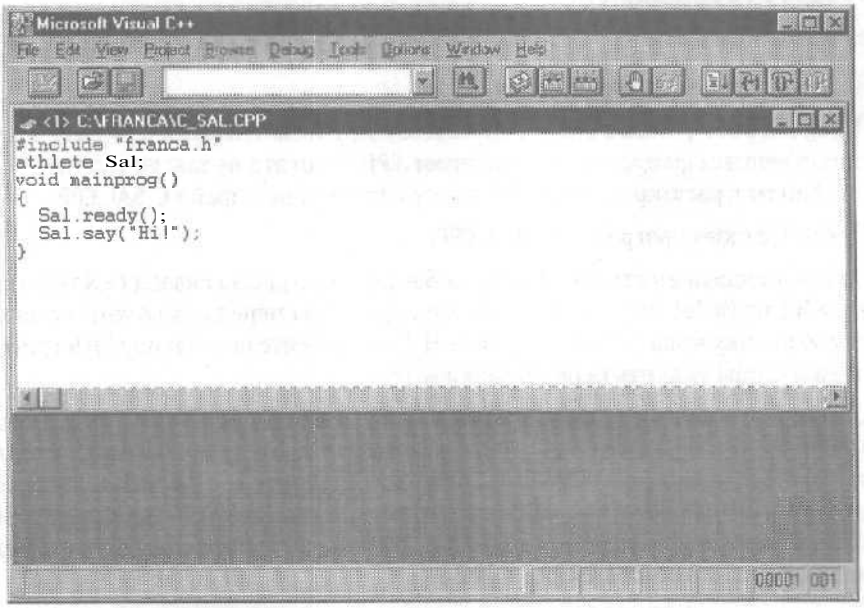


Рис. 0.36. *Открытие файла программы*

Чтобы просматривать и редактировать программу, относящуюся к открытому проекту, дважды щелкните на ее имени. Откроется окно, соответствующее выбранной программе. Когда окно программы откроется, щелкните на том месте, с которого вы хотите начать редактирование, как показано на рис. 0.37.



РМС. ***0.37.*** *Открытие файла C_SAL.CPPв Visual* C++ *1.5*

Окно программы появляется поверх окна проекта. Для перемещения любого окна нажмите левую кнопку мыши на его заголовке и, удерживая ее нажатой, переместите окно. Вы также можете разместить любое окно поверх остальных, щелкнув в любом его месте.

Модифицируя программу, вы утрачиваете ее оригинал. Чтобы избежать этого, откройте окно программы и выберите команду File ➤ Save As. Откроется диалоговое окно Save As, показанное на рис. 0.38. Установите курсор в поле File Name и замените

старое имя (C_SAL.CPP) на новое, которое вы сами выберете (например, NEWSAL.CPP), и затем нажмите кнопку OK.

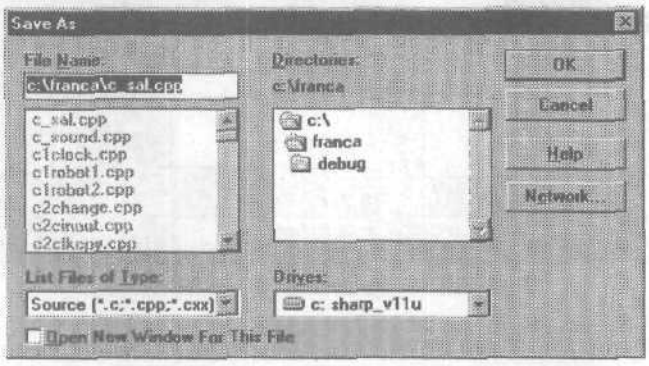


Рис. *0.38.* *Сохранение программы под новым именем*

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Проверьте все файлы в вашем проекте и убедитесь, что файл C_SAL.CPP является единственным файлом с расширением .CPP. Если это не так, то удалите все другие файлы с расширением .CPP и вместо них вставьте файл C_SAL.CPP.
3. Откройте окно программы C_SAL.CPP.
4. На предпоследней строке написано: Sal say («Hi!») (Сэл сказал («Хай!»)); замените Hi! на Hello! (установите ваш курсор справа перед символом H и нажмите левую кнопку мыши. Затем наберите Hello и нажмите на клавишу Delete несколько раз, чтобы удалить первоначальное Hi).
5. Сохраните новую версию программы под именем C_SALNEW.CPP. Убедитесь, что вы сохранили эту профамму в том же каталоге, в котором находятся остальные программы (каталог C:\FRANCA). Если вы выполните теперь ваш проект, как вы думаете, что скажет Сэл? «Hi» или «Hello»? Сэл будет по-прежнему говорить «Hi», поскольку программа C_SALNEW.CPP еще не включена в проект. Если вы мне не верите, запустите программу.
6. Удалите файл C_SAL.CPP из проекта и поместите туда файл C_SALNEW.CPP.
7. Выполните проект. Говорит ли теперь Сэл «Hello»?
8. Удалите файл C_SALNEW.CPP из проекта и верните файл C_SAL.CPP обратно.

Microsoft Visual C++ 4

Если вы используете компилятор Microsoft Visual C++ 4, то следуйте инструкциям этого раздела.

Запуск компилятора

Для запуска компилятора дважды щелкните на значке Microsoft Visual C++ 4. Вы увидите окно, показанное на рис. 0.39.

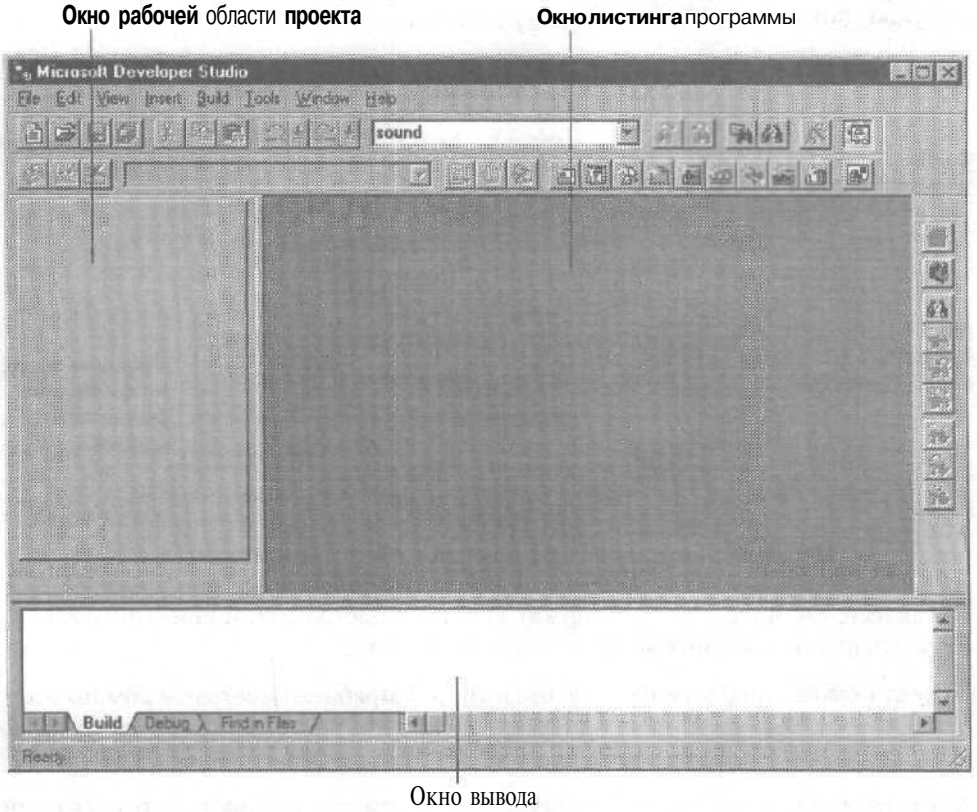


Рис. 0.39. *Окно Microsoft Developer Studio*

Окно Microsoft Developer Studio разделено на три окна:

- Окно рабочей области проекта.
- Окно вывода.
- Окно листинга программы.

Вы можете изменить размер любого окна, перетащив мышью его границу. Вы можете также спрятать любое окно, щелкнув правой кнопкой мыши и выбрав в появившемся контекстном меню команду Hide. Чтобы максимально развернуть окно, щелкните на нем правой кнопкой мыши и выберите в контекстном меню команду Docking View. Если вы свернули какое-либо окно и хотите развернуть его снова, то выберите команду View ➤ Project Workspace или View > Output, в зависимости от того, какое окно вам нужно.

Открытие проекта

В верхней части окна вы видите панель меню.

Выберите команду File ➤ Open Workspace. Откроется диалоговое окно Open Project Workspace, показанное на рис. 0.40.

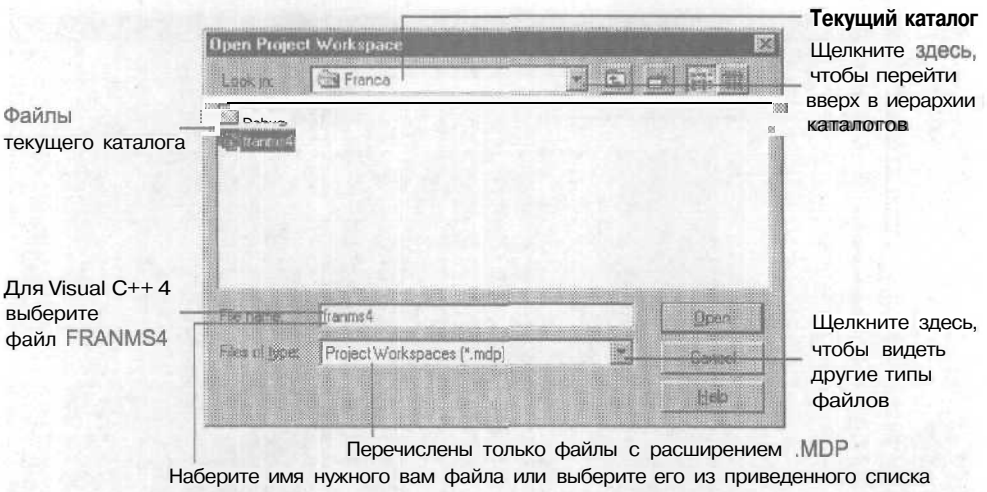


Рис. 0.40, *Диалоговое окно Open Project Workspace в Visual C++ 4*

Убедитесь, что поле Look in содержит каталог с программами книги (C:\FRANCA). Если это не так, выберите из списка нужный каталог.

Проект FRANMS4 подготовлен к работе в среде разработки программ компилятора Visual C++ 4. Выберите проект, щелкнув на нем два раза (вы также можете щелкнуть на проекте один раз, а затем щелкнуть на кнопке OK).

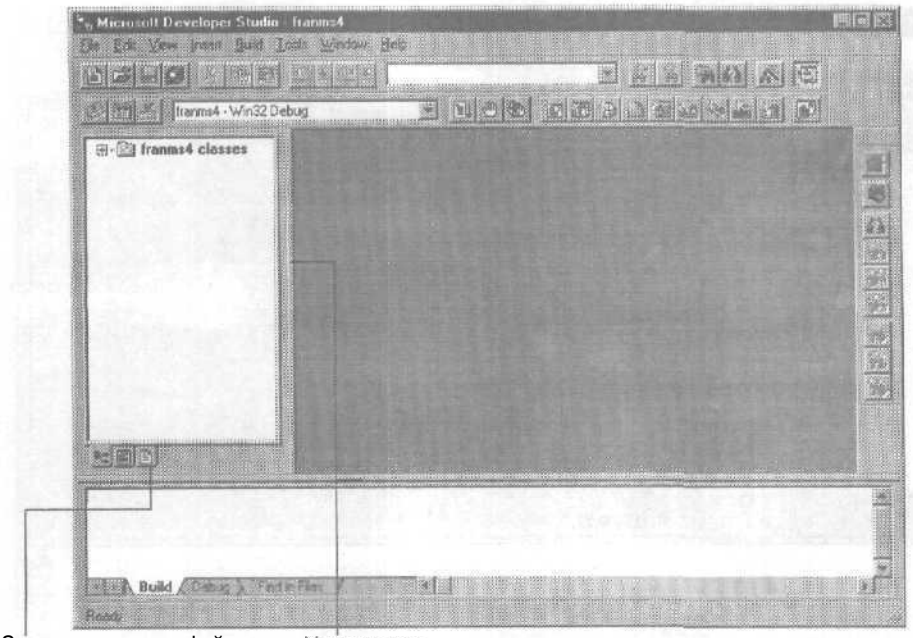
Когда проект открыт, в окне рабочей области проекта появляется проектная информация, показанная на рис. 0.41.

Окно рабочей области проекта по умолчанию отображает информацию об используемых в проекте классах. Вам эта информация пока не нужна. Чтобы увидеть файлы проекта, щелкните на значке просмотра файлов в нижней части окна рабочей области проекта. Теперь экран монитора примет вид, показанный на рис. 0.42.

Пока вы видите единственное имя, представляющее проект. Чтобы увидеть список файлов, щелкните на маленьком квадратике со знаком плюс слева от имени проекта. Немедленно появится список файлов, как показано на рис. 0.43.

Вначале проект будет содержать следующие модули:

- PAULO.RC
- FRANCMS4.OBJ
- C_SAL.CPP



Значок просмотра файлов Увеличивать или уменьшать размер окон можно с помощью мыши

Рис. 0.41. *Открытый проект FRANMS4*

Щелкните здесь, чтобы увидеть список файлов проекта

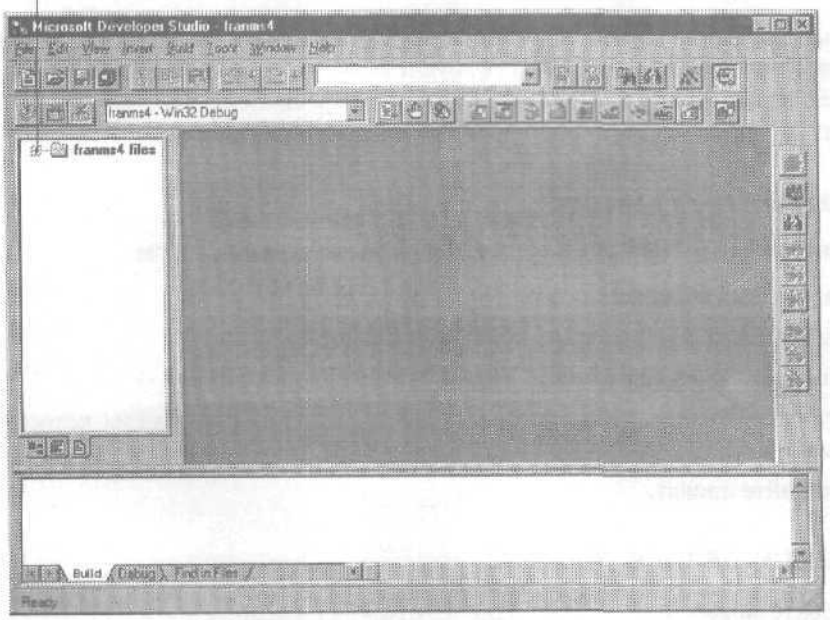


Рис. 0.42. *Окно рабочей области проекта в режиме представления файлов*

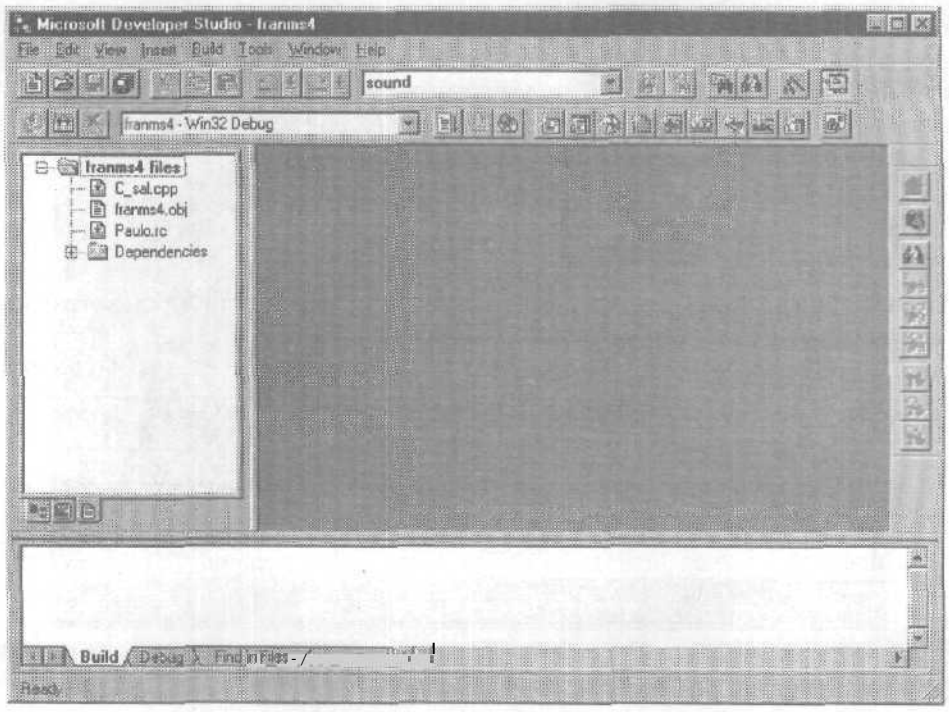


Рис. 0.43. Список файлов проекта

Первые два файла (PAULO.RC и FRANCMS4.OBJ) должны оставаться в вашем проекте всегда. Последний файл (C_SAL.CPP) будет заменяться любой программой, которую вы захотите выполнить. Порядок, в котором перечислены программы значения не имеет.

Проверьте свои навыки!

Убедитесь, что вы поняли, как нужно работать с компилятором.

1. Запустите ваш компилятор.
2. Выберите проект, с которым вы будете работать.
3. Откройте соответствующий проект в вашем компиляторе.
4. Проверьте, какие файлы содержит ваш проект. Те ли это файлы, которые указаны в тексте?
5. Закройте проект.

Редактирование проекта

В проекте может быть только одна программа с расширением .CPP. Это та самая программа на C++, которую вам понадобится выполнять. Все остальные являются

приложениями к ней. На протяжении книги вы будете удалять из проекта одну программу и вставлять на ее место другую. Так вы увидите, как он работает. Исходный проект содержит всего одну программу, которую вы можете просмотреть и которой можете воспользоваться. Чтобы добавлять или удалять файлы, входящие в проект, его сначала нужно открыть.

Удаление программ из проекта

Для удаления файла из проекта выберите его в окне рабочей области проекта, щелкните на нем один раз (при двойном щелчке компилятор выведет листинг программы) и выберите команду Edit ➤ Delete.

ПРИМЕЧАНИЕ Удаляемый файл не стирается с диска. Он просто удаляется из проекта.

ВНИМАНИЕ Удаляйте только файлы с исходными кодами программ. Файлы FRANMS4.OBJ и PAULO.RC должны всегда оставаться в проекте.

Добавление программы в проект

Для добавления программы в открытый проект из главного меню выберите команду Insert ➤ Files into Project, чтобы открыть диалоговое окно Insert Files into Project, показанное на рис. 0.44. Затем выберите нужный вам файл.

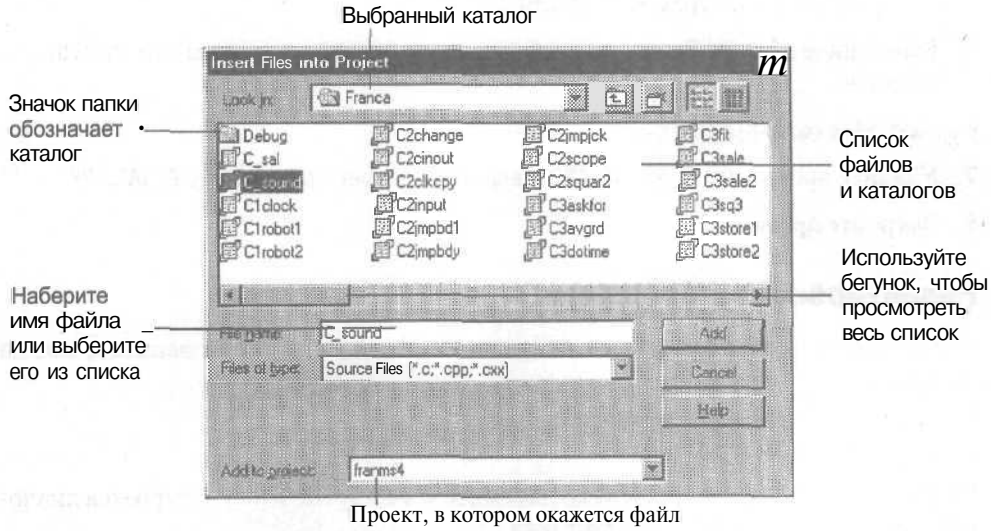


Рис. 0.44. *Выбор файла для добавления в проект*

Выполнение проекта

Чтобы выполнить проект, его надо сначала открыть. Убедитесь, что в вашем проекте имеется одна (и только одна) программа с расширением .CPP. Чтобы выполнить проект, выберите в главном меню команду Build ➤ Debug ➤ Go. Если вы выполняете проект с программой C_SAL.CPP, то увидите экран, аналогичный тому, который показан на рис. 0.8.

После выполнения программы она остается на экране, пока не будет закрыта. Чтобы закрыть окно программы в Windows 3.1, нужно дважды щелкнуть на значке в верхнем левом углу окна. Под Windows 95 для закрытия окна программы нужно щелкнуть на квадратике в правом верхнем углу окна.

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Выполните проект и посмотрите на результат. Действительно ли он выглядит так, как показано на рис. 0.8?
3. Закройте окно программы.
4. Посмотрите на список файлов вашего проекта.

   Удалите программу C_SAL.CPP.

   Вставьте программу C_SOUND.CPP.
5. Выполните проект. Если у вас есть звуковое устройство, то вы что-нибудь услышите.
6. Закройте окно программы.
7. Удалите программу C_SOUND.CPP и верните в проект программу C_SAL.CPP.
8. Закройте проект.

Редактирование программ

Часто бывает необходимо отредактировать существующую или написать новую программу.

Создание новой программы

Чтобы написать новую программу, выберите команду File ➤ New. Откроется диалоговое окно New, показанное на рис. 0.45.

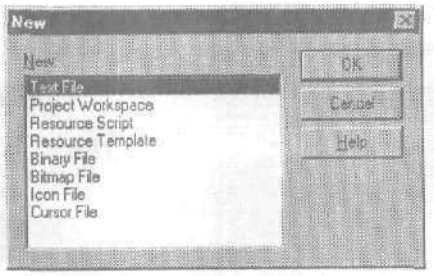


Рис. 0.45. *Выбор текстового файла*

Нажмите кнопку OK, чтобы открылось окно, показанное на рис 0.46. Наберите свою программу и затем выберите команду File ➤ Save As, чтобы сохранить ее на диске. Сохраняя свою новую программу, убедитесь, что выбран именно тот каталог, в который вы хотите ее поместить, и затем наберите новое имя программы в поле File Name.

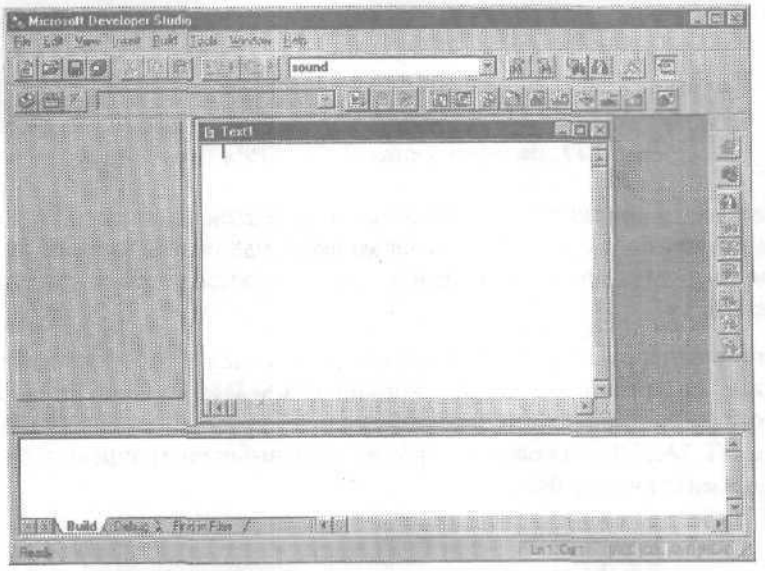


Рис. 0.46. *Окно новой программы*

Редактирование существующей программы

Чтобы просмотреть и отредактировать программу, не входящую в открытый в данный момент проект, выберите команду File ➤ Open. Откроется диалоговое окно Open File. Если вы увидели там нужный вам файл, щелкните два раза на его имени. В противном случае убедитесь, что вы правильно выбрали каталог. Обычно компьютер отображает файлы с расширением .CPP (исходные файлы). Если же вам нужны файлы с другими расширениями, укажите это в данном окне.

Чтобы просматривать и редактировать программу, относящуюся к открытому проекту, дважды щелкните на ее имени. Откроется окно, соответствующее вы-

бранной программе. Когда окно программы откроется, щелкните на том месте, с которого вы хотите начать редактирование, как показано на рис. 0.47.

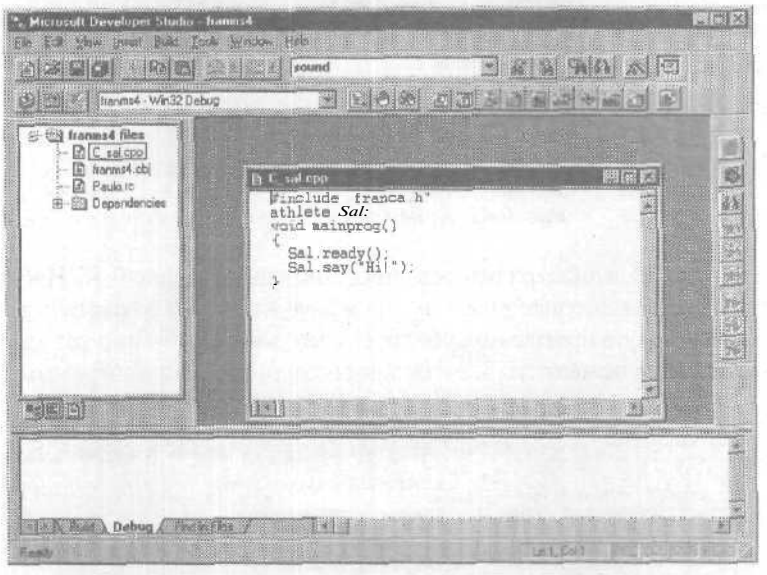


Рис. 0.47. Открытый файл C_SAL.CPP в Visual C++ 4

Окно программы появляется поверх окна проекта. Для перемещения любого окна нажмите левую кнопку мыши на его заголовке и, удерживая ее нажатой, переместите окно. Вы также можете разместить любое окно поверх остальных, щелкнув в любом его месте.

Модифицируя программу, вы утрачиваете ее оригинал. Чтобы избежать этого, откройте окно программы и выберите команду File ► Save As. Откроется диалоговое окно Save As, показанное на рис. 0.48. Установите курсор в поле File Name и замените старое имя (C_SAL.CPP) на новое, которое вы сами выберете (например, NEWSAL.CPP), и затем нажмите кнопку OK.

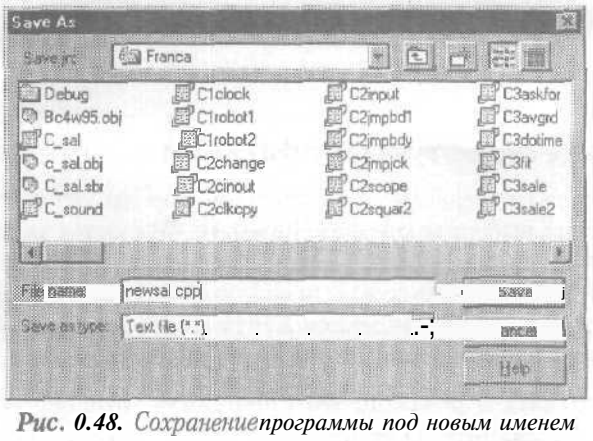


Рис. 0.48. Сохранение программы под новым именем

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Проверьте все файлы в вашем проекте и убедитесь, что файл C_SAL.CPP является единственным файлом с расширением .CPP. Если это не так, то удалите все другие файлы с расширением .CPP и вместо них вставьте файл C_SAL.CPP.
3. Откройте окно программы C_SAL.CPP.
4. На предпоследней строке написано: Sal say («Hi!») (Сэл сказал («Хай!»)); замените Hi! на Hello! (установите ваш курсор справа перед символом H и нажмите левую кнопку мыши. Затем наберите Hello и нажмите на клавишу Delete несколько раз, чтобы удалить первоначальное Hi),
5. Сохраните новую версию программы под именем C_SALNEW.CPP. Убедитесь, что вы сохранили эту программу в том же каталоге, в котором находятся остальные программы (каталог C:\FRANCA). Если вы выполните теперь ваш проект, как вы думаете, что скажет Сэл? «Hi» или «Hello»? Сэл будет по-прежнему говорить «Hi», поскольку программа C_SALNEW.CPP еще не включена в проект. Если вы мне не верите, запустите программу.
6. Удалите файл C_SAL.CPP из проекта и поместите туда файл C_SALNEW.CPP.
7. Выполните проект. Говорит ли теперь Сэл «Hello»?
8. Удалите файл C_SALNEW.CPP из проекта и верните файл C_SAL.CPP обратно.

Microsoft Visual C++ 5

Если вы используете компилятор Microsoft Visual C++ 5, то следуйте инструкциям этого раздела[1].

Запуск компилятора

Для запуска компилятора дважды щелкните на значке Microsoft Visual C++ 4. Вы увидите окно, показанное на рис. 0.49.

Вы можете изменить размер любого окна, перетащив мышью его границу. Вы можете также спрятать любое окно, щелкнув правой кнопкой мыши и выбрав в появившемся контекстном меню команду Hide. Чтобы максимально развернуть окно, щелкните на нем правой кнопкой мыши и выберите в контекстном меню команду Docking View. Если вы свернули какое-либо окно и хотите развернуть его снова, то выберите команду View ➤ Project Workspace или View ➤ Output, в зависимости от того, какое окно вам нужно.

[1] Подробнее о разработке приложений в среде Visual C++ 5 вы можете прочесть в книге С. Холзнера «Visual C++ 5 с самого начала», а если вас интересует новая версия компилятора, вам поможет книга того же автора «Visual C++ 6: учебный курс» (обе книги вышли в издательстве «Питер»).

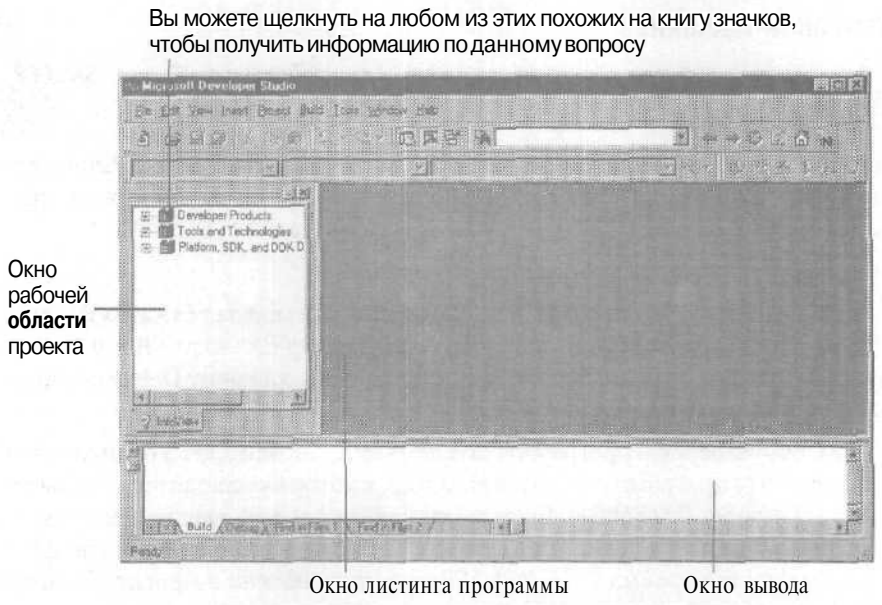


Рис. 0.49. *Окно Microsoft Developer Studio в Visual C++ 5*

Открытие проекта

В верхней части окна вы видите панель меню.

Выберите команду File ➤ Open Workspace. Откроется диалоговое окно Open Workspace, показанное на рис. 0.50.

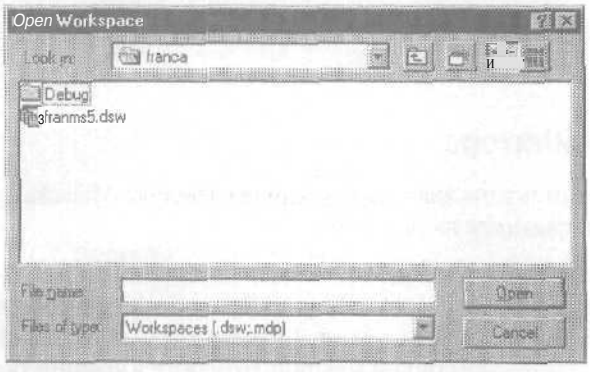


Рис. ***0.50.*** *Диалоговое окно Open Workspace в Visual C++ 5*

Убедитесь, что поле Look In содержит каталог с программами книги (C:\FRANCA). Если это не так, выберите из списка нужный каталог.

Проект FRANMS5 подготовлен к работе в среде разработки программ компилятора Visual C++ 4. Выберите проект, щелкнув на нем два раза (вы также можете щелкнуть на проекте один раз, а затем щелкнуть на кнопке OK).

Когда проект открыт, в окне рабочей области проекта появляется проектная информация, показанная на рис. 0.51.

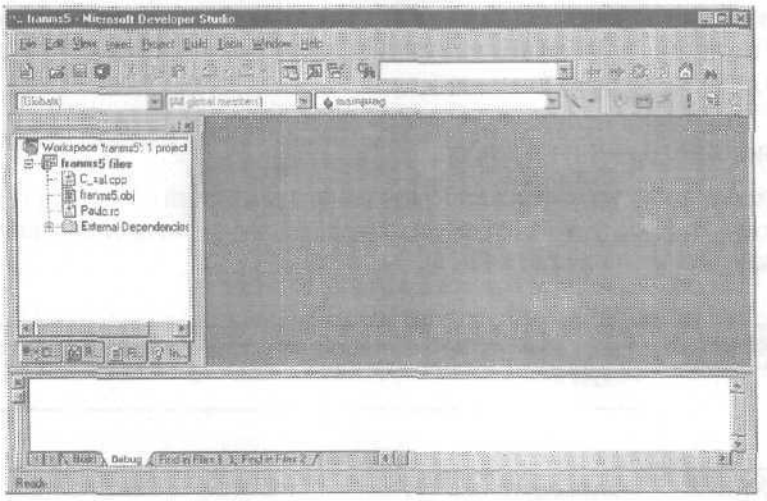


Рис. 0.51. *Окно проекта FRANMS5*

По умолчанию окно рабочей области проекта отображает файлы проекта. Чтобы вывести другую информацию, щелкните на соответствующем значке в нижней части экрана, а пока оставьте окно в прежнем виде.

Вначале проект будет содержать следующие модули:

- PAULO.RC
- FRANMS5.OBJ
- C_SAL.CPP

Первые два файла (PAULO.RC и FRANCMS4.OBJ) должны оставаться в вашем проекте всегда. Последний файл (C_SAL.CPP) будет заменяться любой программой, которую вы захотите выполнить. Порядок, в котором перечислены программы, значения не имеет.

Проверьте свои навыки!

Убедитесь, что вы поняли, как нужно работать с компилятором.

1. Запустите ваш компилятор.
2. Выберите проект, с которым вы будете работать.
3. Откройте соответствующий проект в вашем компиляторе.
4. Проверьте, какие файлы содержит ваш проект. Те ли это файлы, которые указаны в тексте?
5. Закройте проект.

Редактирование **проекта**

В проекте может быть только одна программа с расширением .CPP. Это та самая программа на C++, которую вам понадобится выполнять. Все остальные являются приложениями к ней. На протяжении книги вы будете удалять из проекта одну программу и вставлять на ее место другую. Так вы увидите, как он работает. Исходный проект содержит всего одну программу, которую вы можете просмотреть и которой можете воспользоваться. Чтобы добавлять или удалять файлы, входящие в проект, его сначала нужно открыть.

Удаление программ из проекта

Для удаления файла из проекта выберите его в окне рабочей области проекта, щелкните на нем один раз (при двойном щелчке компилятор выведет листинг программы) и выберите команду Edit ➤ Delete.

ПРИМЕЧАНИЕ Удаляемый файл не стирается с диска. Он просто удаляется из проекта.

ВНИМАНИЕ Удаляйте только файлы с исходными кодами программ. Файлы FRANMS5.OBJ и PAULO.RC должны всегда оставаться в проекте.

Добавление программы в проект

Для добавления программы в открытый проект из главного меню выберите команду Project ➤ Add To Project ➤ Files, чтобы открыть диалоговое окно Insert Files into Project, показанное на рис. 0.52. Затем выберите нужный вам файл. Того же эффекта можно добиться, если щелкнуть правой кнопкой мыши на имени проекта (FRANMS5) и выбрать в контекстном меню команду Add Files To Project.

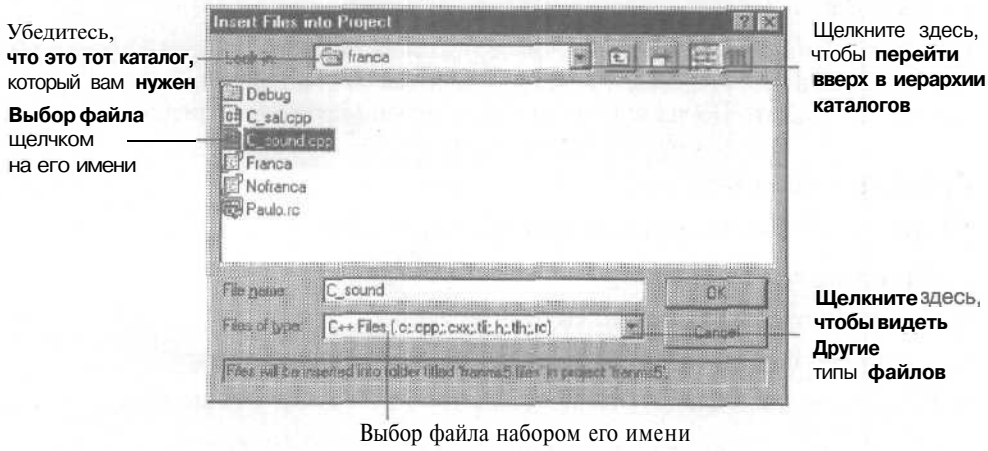


Рис. 0.52. *Выбор файла проекта*

Выполнение проекта

Чтобы выполнить проект, его надо сначала открыть. Убедитесь, что в вашем проекте имеется одна (и только одна) программа с расширением .CPP. Чтобы выполнить проект, выберите в главном меню команду Build ➤ Execute или Build ➤ Start Debug ➤ Go. Если вы выполняете проект с программой C_SAL.CPP, то увидите экран, аналогичный тому, который показан на рис. 0.8.

Перед выполнением проекта необходимо откомпилировать исходную программу (.CPP). Если проект еще не готов, компилятор с помощью окна сообщений выдаст запрос о необходимости перекомпоновки проекта. Нажмите кнопку Yes, а затем снова выберите в главном меню команду Build ➤ Execute или Build у Start Debug ➤ Go.

После выполнения программы она остается на экране, пока не будет закрыта. Чтобы закрыть окно программы в Windows 3.1, нужно дважды щелкнуть на значке в верхнем левом углу окна. Под Windows 95 для закрытия окна программы нужно щелкнуть на квадратике в правом верхнем углу окна.

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Выполните проект и посмотрите на результат. Действительно ли он выглядит так, как показано на рис. 0.8?
3. Закройте окно программы.
4. Посмотрите на список файлов вашего проекта.

   Удалите программу C_SAL.CPP.

   Вставьте программу C_SOUND.CPP.
5. Выполните проект. Если у вас есть звуковое устройство, то вы что-нибудь услышите.
6. Закройте окно программы.
7. Удалите программу C_SOUND.CPP и верните в проект программу C_SAL.CPP.
8. Закройте проект.

Редактирование программ

Часто бывает необходимо отредактировать существующую или написать новую программу.

Создание новой программы

Чтобы написать новую программу, выберите в главном меню команду File ➤ New. Откроется диалоговое окно New, показанное на рис. 0.53.

Нажмите кнопку OK, чтобы открылось окно, показанное на рис 0.54. Наберите свою программу и затем выберите команду File ➤ Save As, чтобы сохранить ее на диске.

Сохраняя новую программу, убедитесь, что выбран именно тот каталог, в который вы хотите ее поместить, и затем наберите новое имя программы в поле File Name.

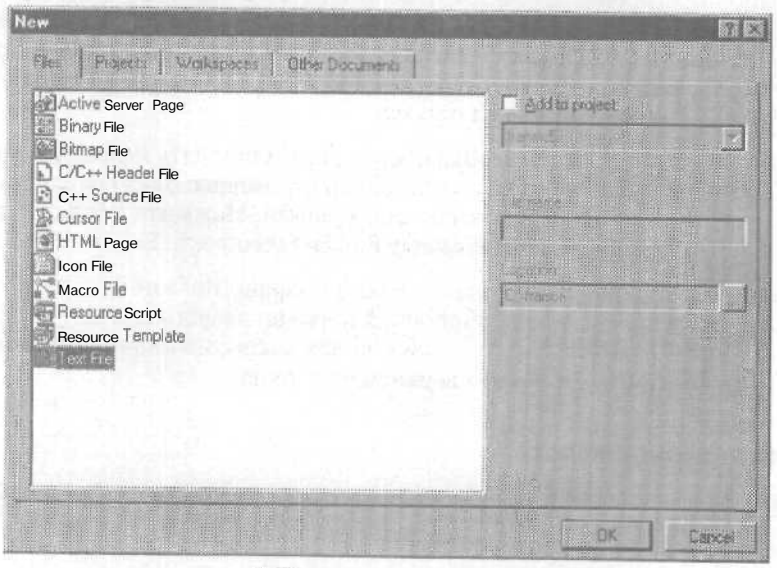


Рис. 0.53. Выбор текстового файла

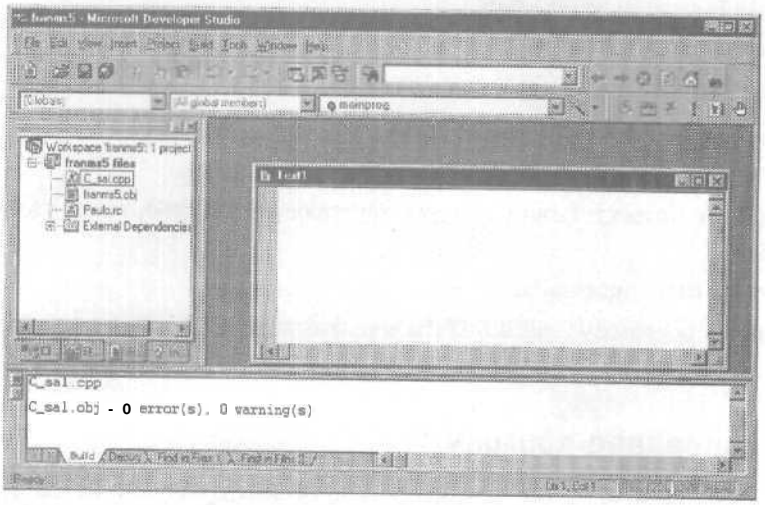


Рис. 0,54. Окно новой программы

Редактирование **существующей** программы

Чтобы просмотреть и отредактировать программу, не входящую в открытый в данный момент проект, выберите команду File ➤ Open. Откроется диалоговое окно File Open. Если вы увидели там нужный вам файл, щелкните два раза на его имени. В противном случае убедитесь, что вы правильно выбрали каталог. Обычно компь-

ютер отображает файлы с расширением .CPP (исходные файлы). Если же вам нужны файлы с другими расширениями, укажите это в данном окне.

Чтобы просматривать и редактировать программу, относящуюся к открытому проекту, дважды щелкните на ее имени. Откроется окно, соответствующее выбранной программе. Когда окно программы откроется, щелкните на том месте, с которого вы хотите начать редактирование, как показано на рис. 0.55.

Учтите, что модифицируя программу, вы утрачиваете ее оригинал. Чтобы избежать этого, откройте окно программы и выберите команду File ➤ Save As. Откроется диалоговое окно Save As, показанное на рис. 0.56. Установите курсор в поле File Name и замените старое имя (C_SAL.CPP) на новое, которое вы сами выберете (например, NEWSAL.CPP), и затем нажмите кнопку OK.

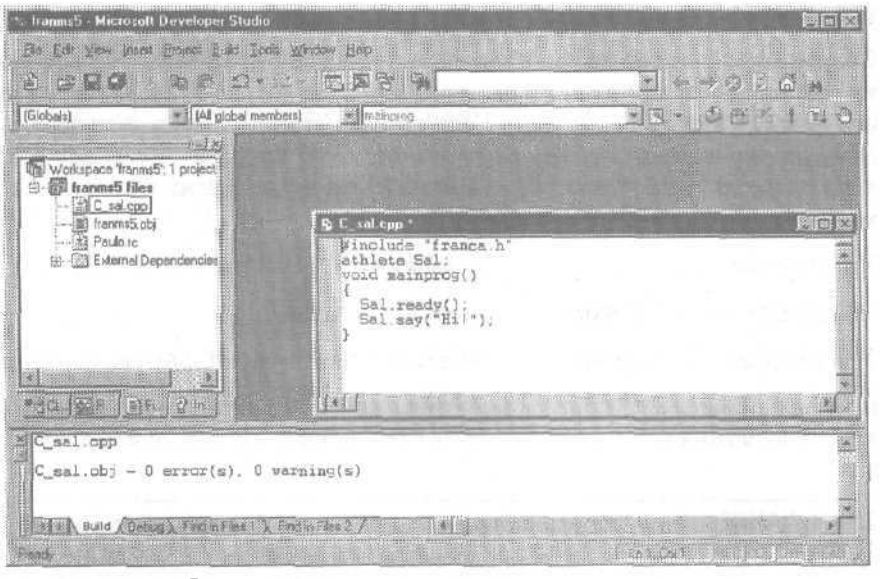


Рис. 0.55. *Файл C_SAL.CPP, открытый в окне Microsoft Visual C++ 5*

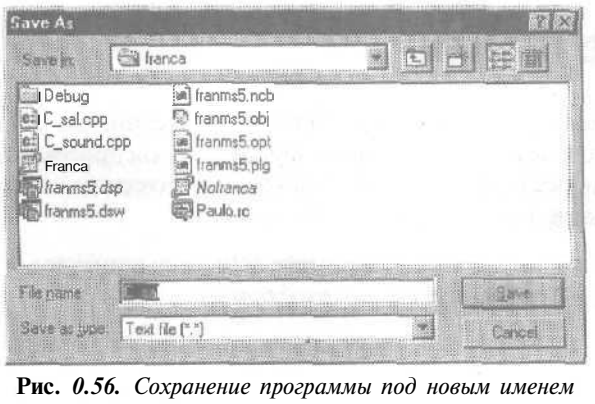


Рис. ***0.56.*** *Сохранение программы под новым именем*

Проверьте свои навыки!

Здесь представлено несколько практических шагов для работы с файлом C_SAL.CPP.

1. Если проект не открыт, откройте его.
2. Проверьте все файлы в вашем проекте и убедитесь, что файл C_SAL.CPP является единственным файлом с расширением .CPP. Если это не так, то удалите все другие файлы с расширением .CPP и вместо них вставьте файл C_SAL.CPP.
3. Откройте окно программы C_SAL.CPP.
4. На предпоследней строке написано: SaL say («Hi!») (Сэл сказал («Хай!»)); замените Hi! на Hello! (установите ваш курсор справа перед символом H и нажмите левую кнопку мыши. Затем наберите Hello и нажмите на клавишу Delete несколько раз, чтобы удалить первоначальное Hi).
5. Сохраните новую версию программы под именем C_SALNEW.CPP. Убедитесь, что вы сохранили эту программу в том же каталоге, в котором находятся остальные программы (каталог C:\FRANCA). Если вы выполните теперь ваш проект, как вы думаете, что скажет Сэл? «Hi» или «Hello»? Сэл будет по-прежнему говорить «Hi», поскольку программа C_SALNEW.CPP еще не включена в проект. Если вы мне не верите, запустите программу.
6. Удалите файл C_SAL.CPP из проекта и поместите туда файл C_SALNEW.CPP.
7. Выполните проект. Говорит ли теперь Сэл «Hello»?
8. Удалите файл C_SALNEW.CPP из проекта и верните файл C_SAL.CPP обратно.

Если вы правильно выполнили все инструкции, а программа все равно не работает, то, вероятно, компилятор не нашел файл FRANCA.H или другой необходимый файл.



ПРИМЕЧАНИЕ Ваш проект должен находиться в том же каталоге, что и все программное обеспечение книги (каталог C:\FRANCA).

Звуковые файлы

Ваши программы могут зазвучать. Для этого достаточно установить звуковой драйвер. Лично я не рекомендую вам озвучивать свои программы, чтобы не создавать лишнего шума в лаборатории! Тем не менее создание собственных звуковых программ может показаться вам интересным.

Звуковые программы хранятся в формате .WAV. Большинство звуковых карт могут записывать звук и сохранять его в этом формате.

Чтобы оценить звуковые способности вашего компьютера, запустите следующую программу:

```
#include "franca.h"
athlete Sal;   //c_sound.cpp
void mainprog()
{
   Sal.ready();
   sound("hello.way");
}
```

Что нового мы узнали?

В этом уроке мы научились

- Понимать проблемы программирования.
- Открывать проекты и менять в них программы.
- Модифицировать программы.
- Создавать новые файлы программ.
- Сохранять файлы программ.
- Запускать с помощью компилятора примеры программ из программного обеспечения книги.

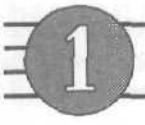
Часть I

Ваши первые программы

В части I вы научитесь понимать и разрабатывать простейшие программы. К этому моменту вы уже должны уметь пользоваться компилятором C++, описанным в части 0.

Компьютерные программы представляют собой инструкции, объясняющие компьютеру, как решать ту или иную задачу. Простейшие типы задач решаются простой последовательностью действий. В этих последовательных программах нет проверки условий, а также отсутствуют какие бы то ни было повторения (циклы). Именно такие программы мы рассмотрим в части I.

Изучив представленный материал, вы сможете понимать, модифицировать и создавать простейшие программы, а также организовывать диалог с пользователем и решать простые задачи.



УРОК 1 Разработка и модификация программ

- Программные инструкции
- Отправка сообщений объектам
- Аргументы
- Использование объектов
- Комментарии

Этот урок поможет вам уяснить, как пишутся простые компьютерные программы. В нескольких приводимых программах мы снова встретимся с Сэлом, знакомым нам по части 0. Вы сможете увидеть его на экране, где он выполнит кое-какие упражнения и «произнесет» несколько слов.

Первая программа

Мы начнем с простой программы, которую вы легко сможете выполнить на своем компьютере. Чтобы понять, как функционируют компьютерные программы, попытайтесь сразу же ее запустить. Для этого воспользуйтесь проектом, который позволяет создать ваш компилятор.



ПРИМЕЧАНИЕ О том, как правильно запускать программы в среде разработки вашего компилятора, можно узнать из материала урока 0.

Программа C_SAL.CPP выводит на экран картинку. Ниже представлен код этой программы:

```
#include "franca.h"
athleteSal
void mainprog()
{
   Sal.ready();
   Sal.say("Hi!");
}
```

Рассмотрим каждую строку программного кода.

tfincLude — эта строка сообщает компьютеру, что он должен вставить в это место другие строки, которые я, как автор, подготовил для включения в ваши программы. Дополнительные строки содержат инструкции, позволяющие сделать ваши программы проще и интереснее. Компьютер находит эти строки по имени файла franca.h. На самом деле эта строка не относится к языку C++. Она указывает компилятору (точнее его составной части, называемой *препроцессором)* найти эти заранее подготовленные части программы и вставить их в программу в этом месте.

Таким образом, появляется возможность вставлять в ваши программы части других программ, разработанных вами или кем-то еще. А это реальная экономия времени! Некоторые действия, которые нам раз за разом приходится выполнять, благодаря компилятору могут быть собраны воедино (программы, реализующие эти действия, можно либо купить, либо позаимствовать у ваших коллег). Для присоединения таких программ к нашим программам и используется директива tfinclude.

athlete Sal — в ваших первых программах для создания изображений мы будем использовать специальный объект. В файле franca.h задается особый тип или *класс* объектов, который мы назвали athlete (гимнаст).

Нашего гимнаста зовут Сэл (Sal). Таким образом, эта строка всего-навсего объясняет компилятору, что Sal — это объект класса athlete. Поскольку объекты класса athlete могут изображаться в разных позициях, то тоже самое относится и к Сэлу.

void mainprog() — начиная с этой строки идет описание основной последовательности действий, которую вы задаете своему компьютеру. Эти действия состоят из инструкций, заключенных между открывающей и закрывающей фигурными скобками ({}), показанными на следующей строке и несколькими строками ниже.



ПРИМЕЧАНИЕ В C++ фигурные скобки ограничивают последовательность действий.

На самом деле основная последовательность действий в C++ обозначается функцией main() или winmain(). Имя mainprog() используется только в программном обеспечении книги. В следующих уроках об этих различиях будет рассказано более подробно.



ПРИМЕЧАНИЕ Использование функции mainprogQ для обозначения основной функции программы — это просто специальный трюк, призванный помочь обучению. Реальная функция winmain() встроена в программное обеспечение книги и вызывается через функцию mainprog(). Именем winmain() система называет программу, обеспечивающую связь с графической средой Windows. Но в наших программах, вплоть до части VIII, основной функцией будет функция mainprog().

Sal.ready() — это строка указывает Сэлу подготовиться к выполнению упражнений. Для этого мы посылаем сообщение ready объекту Sal. Обычный формат пересылки сообщения состоит из имен объекта (Sal) и сообщения (ready), между которыми ставится точка, а следом за именем сообщения располагаются круглые скобки (()).

Sal.say("Hi!") — это другое сообщение, посылаемое Сэлу. Мы хотим, чтобы Сэл сказал «Hi» (привет). Действительно ли Сэл будет говорить? Ну, конечно, не в прямом смысле, но уж по крайней мере вы получите от него сообщение! Заметьте, что

в этом случае внутри скобок находится то, что Сэл должен сказать. Слова Сэла заключены в двойные кавычки.

Если хотите, можете выполнить эту программу прямо сейчас. Запустите ваш компилятор и откройте соответствующий проект. Проверьте программу и выполните ее! Результат показан на рис. 1.1.

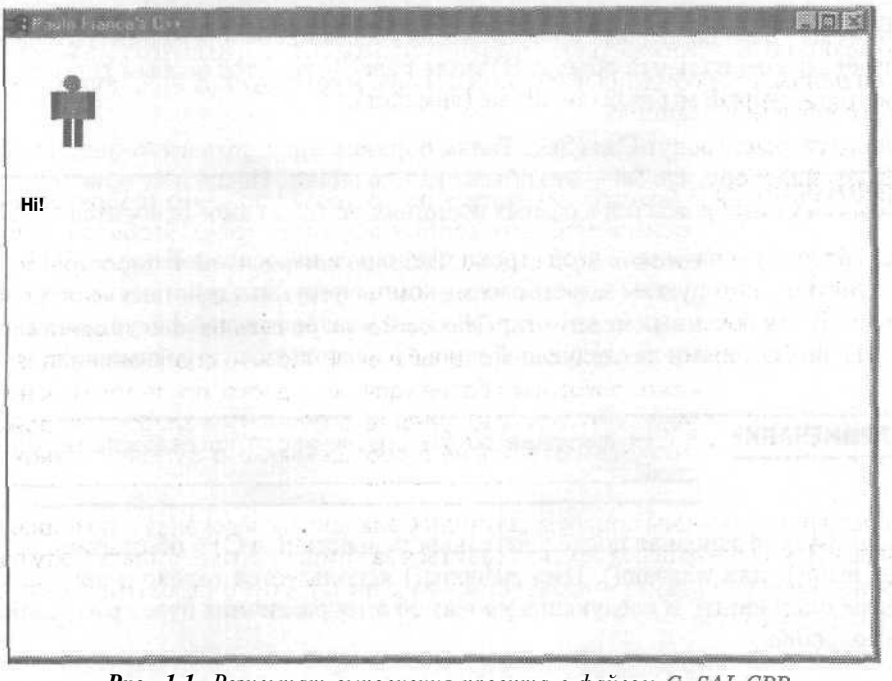


Рис. 1.1. *Результат выполнения проекта с файлом C_SAL.CPP*



СОВЕТ

Если вы еще не прочитали урок 0, сделайте это сейчас. Вы узнаете, как выполнить описанные в книге программы на вашем компиляторе. В уроке 0 представлены некоторые наиболее популярные компиляторы C++.

Отправка сообщений

Теперь, когда вы знаете, как запускать программы, пора приступить к их написанию. Вы будете приказывать компьютеру работать с так называемыми *объектами* (objects).

Чтобы работать было проще и интереснее, я приготовил для вас несколько объектов. С объектом Sal, который будет нашим компаньоном на первых уроках, вы уже встречались. Этот объект можно наблюдать на экране в нескольких позициях:

ready (готов)

up (руки вверх)

left (руки влево)

right (руки вправо)

При отправке объекту Sal соответствующего сообщения он будет занимать требуемую позицию. Вы уже наверно догадались, что мы создали нашего гимнаста таким, что он понимает сообщения ready, up, left и right и реагирует на них, занимая заданную в сообщении позицию.



ПРИМЕЧАНИЕ В базовой литературе по программированию говорится, что мы посылаем объекту сообщение, когда хотим, чтобы он выполнил какое-либо действие. В более специальной литературе по C++ пишут, что для этого мы используем *функцию-член* (member function) объекта. Оба описания по сути означают одно и то же. На начальном этапе обучения термин сообщение (message) может показаться более удачным, однако, после того как на последующих уроках вы узнаете о функциях и классах, то поймете, что лучше говорить не о сообщениях, а о функциях-членах.

В первой программе мы попросту научили Сэла занимать исходную позицию и говорить «Hi». Однако перед тем, как садиться за компьютер и посылать Сэлу разного рода сообщения, нужно познакомиться с несколькими базовыми понятиями программирования.

Инструкции

Инструкции (statements) — это указания вашему компьютеру. Некоторые инструкции представляют собой сообщения объектам. *Сообщения*— это понятные объектам инструкции. Компьютер (и его объекты) единовременно может выполнить только одну инструкцию. В одной строке можно написать одну за другой несколько инструкций, но программисты давно поняли, что гораздо проще и понятнее размещать каждую инструкцию в отдельной строке.

Можно использовать только те сообщения, которые понимает объект. В нашем случае это сообщения ready, up, Left и right. Если вы попросите Сэла, например, лечь на пол (сообщение to go down), то он этого не сделает, поскольку не знает, что означает такое сообщение!



ПРИМЕЧАНИЕ Объекты класса athlete соответствуют этим специальным сообщениям (иначе говоря, в класс включены только эти функции-члены).

На рис. 1.2 показаны все возможные действия Сэла.

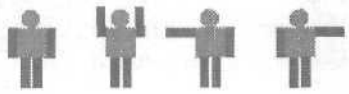


Рис. 1.2. *Все инструкции, которые понимает класс Sal (пока)*

Вы должны объявлять, какому объекту предназначено ваше сообщение. Некоторые программы могут работать с несколькими объектами. Например, у вас могут быть два гимнаста: Сэл и Салли. Если вы просто передадите сообщение up, то, как вы думаете, кто из них поднимет руки? Сэл? Салли? Оба? Чтобы идентифицировать объект, перед сообщением необходимо указывать имя объекта, как это сделано ниже:

```
Sal.ready();
```

Вы должны дать знать компьютеру, какие объекты вы используете и к какому классу они принадлежат.

Компилятор C++ различает прописные и строчные буквы. Имя Sal — это не то же самое, что sal, поэтому при написании программ вы должны учитывать *чувствительность к регистру символов,*

ВНИМАНИЕ Убедитесь, что ваши команды и имена классов остаются в программе неизменными. Использование в одном месте имени Sal, а в другом sal ведет к ошибке.

Каждая инструкция в вашей программе должна оканчиваться точкой с запятой (имеется ряд исключений, например инструкции include и void main()).

Чтобы послать объекту сообщение, нужно последовательно написать имя объекта, точку, само сообщение и далее открывающую и закрывающую скобки (впоследствии внутрь этих скобок мы научимся что-нибудь вставлять). Например:

```
Sal.readyO;
```

Эта инструкция предписывает Сэлу занять позицию готовности.

Жучки

Вы уже могли обратить внимание, что даже малейшая ошибка (вроде неправильного написания имени, пропуска точки и т. д.) заставляет компьютер выдавать сообщение об ошибке или работать неправильно (если вы еще в этом не убедились, не отчаивайтесь — вы встретитесь с этим гораздо раньше, чем думаете). Это позволяет лишний раз убедиться, что компьютер всегда выполняет только то, что ему приказано.

Если компьютер делает что-то не так, то это происходит только потому, что вы ему так объяснили! Иногда вам кажется, что компьютер сошел с ума, но если вы внимательно просмотрите свои инструкции, то наверняка обнаружите ошибку. На программистском жаргоне такие ошибки называются *жучками* (bugs).

Класс athlete

Вы можете объявлять объекты класса athlete и использовать их в своих программах. Все объекты этого класса имеют характеристики, проиллюстрированные на рис. 1.2.

Каждый объект занимает на экране квадратную ячейку. Первый объект занимает левую верхнюю ячейку, а каждый объявляемый вами объект занимает следующую направо ячейку.

Форма этих объектов соответствует сообщениям:

```
ready
up
left
right
```

Вдобавок гимнасты умеют «говорить». Чтобы это случилось, пошлите гимнасту сообщение say (заключив текст сообщения в круглые скобки и парные кавычки).

Аргументы

Аргументы (arguments) позволяют расширить рамки сообщений. Можно, например, изменять скорость, с которой работают гимнасты, задавая число секунд, которое гимнаст должен находиться в той или иной позиции. Сообщение say тоже имеет аргумент, определяющий то, что произносит гимнаст.

Движения могут выполняться быстрее или медленнее. Чтобы ввести в сообщение требуемую скорость, в качестве аргумента внутри скобок нужно просто указать число секунд. Например, следующие две строки предлагают Сэлу оставаться 5 секунд с поднятой левой рукой, а затем на 8 секунд поднять обе руки:

```
Sal.left(5);
Sal.up(8);
```

Аргументы, соответствующие сообщениям ready, up, left и right, могут быть дробными. Например,

```
Sal.left(0.5);
```

Если вы не указали аргумент, то по умолчанию он считается равным одной секунде. Если в качестве аргумента вы используете ноль, то компьютер выполнит операцию максимально быстро. (Возможно, вы даже не успеете это заметить!)

В сообщениях ready, up, left и right используются численные аргументы, то есть числа, обозначающие значение промежутка времени, С другой стороны, в сообщении say можно указывать как числа, так и последовательность из букв, цифр и знаков, которая выводится на экран в полном соответствии с тем, как она задана внутри парных кавычек.



ВНИМАНИЕ Если вы сохраняете модифицированную программу, то можете потерять оригинал! Чтобы этого не случилось, выберите команду File ➤ Save As и дайте файлу новое имя.

Почему, когда мы хотим заставить Сэла что-то сказать, мы заключаем это «что-то» в парные кавычки? Допустим, вы хотите чтобы Сэл сказал «hello»:

```
Sal.say(hello);
```

Это сообщение вызывает проблему, поскольку аргумент hello может оказаться именем какого-нибудь объекта и ваш компьютер не поймет, что вы от него хотите: вывести на экран слово «hello» или содержимое некоторого объекта hello. Таким образом, кавычки сообщают компьютеру, что вы хотите вывести на экран в точности то, что находится внутри кавычек.



ПРИМЕЧАНИЕ Работая в C++, для обозначения выводимого на экран текста вы всегда должны использовать кавычки. Компиляторы C++ различают аргументы hello и "hello".

Оформление программ на C++

Хороший программист всегда стремится, чтобы его программы были понятны другим. Поэтому никогда не считайте, что работа над программой заканчивается вместе с ее запуском. Вы или кто-то другой может снова воспользоваться вашей программой или же попытается ее модифицировать. Более того, возможно вам самому придется разбирать и модифицировать программы, написанные кем-то другим! В любом случае крайне важно, чтобы не только вы, но и другие могли разобраться в ваших программах.

Одним из способов, облегчающих понимание ваших программ, являются *комментарии*. Комментарии игнорируются компьютером и не влияют на работу программы, их назначение — всего лишь пояснять ваши действия другим. (Впоследствии они помогут и вам вспомнить, что и зачем вы делали.)

Возможно, сейчас у вас нет особого желания вставлять комментарии в программы, поскольку последние еще невелики и, я надеюсь, понятны, но не попадайтесь на эту удочку! Объем и сложность ваших программ будут все время расти. Начните использовать комментарии уже сейчас!

В C++ перед комментариям ставится двойная косая черта (//), весь текст за которой в строке рассматривается как комментарий.

Например, после добавления комментариев наша первая программа будет выглядеть следующим образом:

```
//*********************************************
//            Программа C_Sal.CPP
// Эта программа показывает на экране гимнаста
// Составлена Paulo Franca,  21.08.94
// Последняя доработка 21.08.94
//*********************************************
#include "franca.h"   // Использование программного обеспечения книги
athlete Sal;          // Объявление Сэла в качестве гимнаста
void mainprog()       // Начало программы
{
   Sal.ready();       // Приказываем Сэлу встать в позицию
                      // готовности
   Sal.say("Hi!");    // Приказываем Сэлу сказать "Hi!"
}                     // Конец программы
```

После двойной косой черты можно писать все что угодно. Если текст не помещается на одной строке, просто воспользуйтесь следующей, как это сделано после инструкции Sal.ready(). Комментарии могут находиться как в начале строки, так и в любом другом месте по вашему выбору. Главное, не забывайте о них!



СОВЕТ Помимо поясняющих комментариев, всегда полезно в начале программы указать имя автора программы, даты ее написания и последнего изменения, а также краткое описание программы.

Отметьте, что все комментарии сдвинуты вправо. Так проще определить, где начинаются и где оканчиваются действия. Используйте такую же структуру для записи программных инструкций (обратите внимание, что внутри фигурных скобок все инструкции также сдвинуты вправо).

Понятие последовательности

Чтобы выполнить задачу, вы должны объяснить компьютеру все этапы его работы. Естественно, для этого задачу нужно понимать.

Алгоритм — это точное и подробное описание выполнения задания, которое в принципе может быть написано на любом языке. *Программа* — это алгоритм, записанный в понятной компьютеру форме.

Пока мы будем использовать только *последовательные* алгоритмы, выполняющие действия последовательно, одно за другим. На следующих уроках мы научимся включать в алгоритмы циклы и условия.

Последовательные шаги

Если вы можете составить рецепт приготовления яичницы или же описать, как добраться до ближайшей станции метро, то наверняка сумеете объяснить компьютеру, как решить ту или иную задачу. Необходимо только помнить две вещи:

- Невозможно объяснить то, что не знаешь сам. Вы никогда не объясните, где находится станция метро, если сами этого не знаете. Не важно, кому вы объясняете, человеку или компьютеру, убедитесь, что знаете, о чем говорите.
- Для компьютера ничего не очевидно заранее. Вдобавок он почти нем и у него полностью отсутствует здравый смысл. Если вы не объясните ему все подробно, он просто «зависнет».

Инструкции Сэлу

Начнем с того, что заставим Сэла выполнить несколько упражнений. Например, команды:

```
ready
up
ready
```

Смогли бы вы написать программу, позволяющую Сэлу выполнить эти команды? Единственное, что вы можете, это посылать Сэлу соответствующие сообщения. Можно просто взять исходную программу и вставить в нее новые сообщения. Пробуйте! Но при этом помните о следующем:

- На каждой строке может быть только одна инструкция.
- После инструкции ставится точка с запятой.
- Убедитесь, что в сообщение Сэлу входит имя объекта, точка, имя сообщения и круглые скобки. Например, `Sal.right()`, `Sal.left()` и т. д.

Если вы правильно выполните упражнение, то увидите, как Сэл поднимает руки вверх, а затем опускает их вниз.

Хотите попробовать еще? А как насчет следующей последовательности сообщений:

```
ready
up
left
```

```
up
ready
up
right
up
ready
```

Если вы правильно составите эти программы, то увидите, как упражняется Сэл. Перед тем как двигаться дальше, попробуйте выполнить и другие упражнения. Я хочу, чтобы вы уяснили себе характер работы компьютера, который последовательно просматривает и выполняет ваши инструкции.

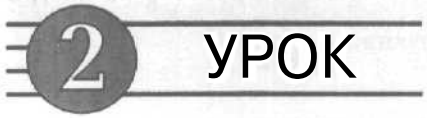
Самостоятельная практика

- Измените предыдущую программу так, чтобы выполнение упражнений стало медленнее. Для этого вставьте число, например 5, внутрь скобок соответствующих сообщений ready, up, left, right.
- Измените программу так, чтобы ускорить выполнение упражнений. Для этого вставьте в скобки 0.
- Измените программу C_SAL.CPP так, чтобы Сэл говорил «Sal» вместо «Hi!».
- Объявите в дополнение к объекту Sal еще один объект, например Sally (Салли), и пусть каждый объект сообщит свое имя.
- Заставьте Сэла, а затем и Салли выполнить следующие упражнения: ready, up, left, up, right, up, ready.
- Заставьте Сэла по ходу выполнения упражнений говорить, какое упражнение он делает (ready, up, left, up, right, up, ready).

Что нового мы узнали?

В этом уроке мы научились

- Идентифицировать инструкции в программах на C++.
- Посылать объектам различных типов сообщения, включая строки (например, «Hello!») и числа.
- Составлять последовательные алгоритмы.
- Использовать объекты класса athlete.
- Пользоваться комментариями.



Вывод информации на экран

- Объекты класса Clock (часы)
- Объекты класса Box (рамка)
- Объекты класса Robot (робот)

На этом уроке вы узнаете, как определять время работы ваших программ, а также научите компьютер выдерживать определенную паузу. В этом вам помогут объекты класса Clock.

Кроме того, здесь вы научите компьютер общаться с вами или любым другим пользователем. Часто бывает необходимо выводить на экран некоторую информацию, отображающую ход выполнения программы или итог ее работы. Для этих целей служат объекты класса Box, в которых можно выводить значения или сообщения- Каждый такой объект может быть снабжен меткой (label), идентифицирующей его назначение.

Чтобы разрабатывать более интересные программы, вы познакомитесь с еще одним классом объектов, классом Robot. С помощью объектов этого класса удобно рисовать на экране простые изображения. (Впоследствии мы воспользуемся ими, чтобы проиллюстрировать решение более сложных проблем.)

Объекты типа Clock

Объявляя объекты типа Clock (или класса Clock, как вам больше нравится), мы получаем возможность контролировать время в наших программах. Для объявления объекта типа Clock, как и для объявления любого другого объекта программы, ему назначается идентификатор:

```
Clock mywatch;    // mywatch  - это объект типа Clock
```

После того как объект объявлен, его можно использовать для следующих целей:

- Определять время, прошедшее с момента объявления объекта.
- Приостанавливать выполнение программы на заданный промежуток времени.

В любой момент времени вы можете перезапустить свои часы.

Сообщения объектам типа Clock

Объекты типа Clock могут реагировать на следующие сообщения:

```
wait(число секунд);
time();
reset();
```

Сообщение wait (число секунд) приостанавливает работу компьютера на число секунд, указанное в скобках в качестве аргумента. Это число может быть дробным.

Сообщение time() заставляет объект типа Clock показать время (в секундах), прошедшее с момента создания или инициализации этого объекта (позже вы поймете, как это можно использовать).

Сообщение reset() возвращает объект типа Clock в исходное состояние. При объявлении показания всех часов автоматически сбрасываются в 0, поэтому перед использованием объектов типа Clock нет необходимости специально возвращать их в исходное состояние.

Использование объектов типа Clock

Допустим, что с помощью часов мы хотим сначала заставить Сэла сказать: «Hello!», а спустя 3 секунды произнести: «How are you?». Для этого нужно сделать следующее:

1. Послать сообщение Сэлу с аргументом "Hello!".
2. Послать сообщение объекту класса Clock с требованием подождать 3 секунды.
3. Послать сообщение Сэлу с аргументом "How are you?".

Но в первую очередь нужно объявить объекты класса athlete (это объект Sal) и класса Clock. Кроме того, чтобы Сэл появился на экране, ему нужно послать какое-нибудь сообщение (например, ready), иначе он останется невидимым!

В приведенной ниже программе C1CLOCK.CPP роль часов играет объект mywatch.

```
// Эта программа иллюстрирует работу часов
// Составлена Пауло Франка 19.10.94.
//          C1CLOCK.CPP
#include "franca.h"
void mainprog()
{
   Clock mywatch;
   athlete Sal;
   Sal.ready();                  // Вывод Сэла на экран
   Sal say("Hello!");           // Сэл произносит "Hello!"
   mywatch.wait(3);              // Перед тем как сделать следующий шаг,
                                 // Сэл ждет 3 секунды
   Sal.say("How are you?");     // Теперь Сэл произносит "How are you?"
   mywatch.wait( 1);             // Сэл ждет 1 секунду
}
```

Аргументом объектов класса Clock являются числа, которые могут быть дробными. Время, возвращаемое в ответ на сообщение time(), также может содержать дробную часть. Таким образом, с помощью следующей инструкции вы можете задать время ожидания, например, в полсекунды:

```
mywatch.wait(0.5);
```

В начале работы программы объект mywatch автоматически сбрасывается в ноль. Следовательно, мы всегда можем вывести на экран время в секундах, прошедшее с момента запуска программы. Для этого используется следующая инструкция:

```
Sal.say(mywatch.time());
```

В этом случае мы увидим время, реально прошедшее с момента запуска программы, поскольку сообщение mywatch.time() заставляет объект типа Clock возвратить именно это значение, а единственный доступный нам пока способ узнать какое-либо значение — это заставить Сэла назвать его.

С чего начинается создание программы?

Необходимость создания новой программы иногда может вызвать у вас чувство полной беспомощности. Не зная, с чего начать, вы просто с благоговейным трепетом будете глядеть на экран своего компьютера. Это обычный симптом начинающих. Я рекомендую выработать привычку записывать для себя все необходимые действия. В дальнейшем этот список можно использовать в качестве комментариев к программе. Такой образ действий будет способствовать организации вашего мышления и поможет разобраться в проблеме.

Например, пусть вы хотите, чтобы на экране появлялся Сэл и произносил: «Sally! Where are you?» («Салли! Где ты?»), а затем через 5 секунд на экране появилась бы Салли и отвечала: «Here I am!» («Я здесь!»).

К сожалению, наши герои могут общаться только очень короткими фразами, поскольку в соответствующих информационных рамках (объектах типа Box) им отведено не слишком много места. Однако вы можете заставить их говорить и длинными фразами, разбивая фразы на отдельные части. Например, вместо того чтобы сразу сказать: «Sally! Where are you?», Сэл может произнести эту фразу в несколько приемов с двухсекундным интервалом между словами. Если вы последуете моему совету и начнете комментировать выполняемые действия, то у вас получится следующее:

```
// Объявление двух атлетов (объектов класса athlete)
// Вывод Сэла на экран
// Сэл говорит "Sally! Where are you?",
//    пропуская по 2 секунды между словами
// Пауза в 5 секунд
// Вывод Салли на экран
// Салли говорит "Here I am!",
//    пропуская по 2 секунды между словами
```

Эти комментарии достаточно точно описывают алгоритм выполнения программы. Внимательно просмотрев их, вы, возможно, сразу сумеете приступить к написанию программы. Если же нет, вносите в них дополнительные комментарии, пока содержание будущей программы не станет для вас совершенно очевидным.

Вам может не понравиться комментарий, в котором говорится, что Сэл делает двухсекундные паузы между словами. Возможно, вы захотите пояснить это лучше. Нет проблем — просто замените этот комментарий на более подробный:

```
// Сэл говорит "Sal ly! "
// Пауза в 2 секунды
// Сэл говорит "Where"
// Пауза в 2 секунды
// Сэл говорит "are you?"
```

Вставьте эти комментарии на место, и ваша программа примет следующий вид:

```
// Объявление двух атлетов (объектов класса athlete)
// Вывод Сэла на экран
//   Сэл говорит "Sal ly! "
//    Пауза в 2 секунды
//   Сэл говорит "Where"
//    Пауза в 2 секунды
//    Сэл говорит "are you?"
// Пауза в 5 секунд
// Вывод Салли на экран
// Салли говорит "Here I am! ",
//    пропуская по 2 секунды между словами
```



ПРИМЕЧАНИЕ Отступы поясняют, что эти действия образуют кок бы одно комплексное действие (заставляют Сэла произнести фразу полностью). Точно так же можно написать аналогичные комментарии для Салли.

Добавление инструкций к комментариям

Теперь, когда вы располагаете хорошим описанием вашей программы, к комментариям можно добавить настоящие инструкции C++. Например, непосредственно под строкой:

```
// Объявление двух атлетов (объектов класса athlete)
```

можно разместить следующую инструкцию:

```
athlete Sal, Sally;
```

Добавьте все инструкции — и программа готова. Привыкните описывать этапы выполнения вашей программы в качестве комментариев, а потом добавлять к ним инструкции. Это особенно полезно, когда не знаешь, с чего начать,

В качестве упражнения попытайтесь закончить и выполнить нашу программу.

Объекты типа Box

Допустим, что в какой-то из предыдущих программ вы захотите вывести на экран информацию о том, сколько времени занимает выполнение программы. Эту информацию может легко предоставить объект класса Clock, но как вывести ее на экран?

Единственный способ, который нам доступен к настоящему моменту, — это заставить Сэла «назвать» временной интервал. Если же вы хотите вывести эту информацию каким-то другим способом, то вам понадобятся другие объекты.

Объекты класса Box аналогичны объектам класса athlete. Объявив единожды объект класса Box, вы можете заставить его «говорить» все, что захотите, точно так же, как это было с объектами класса athlete. Однако объекты класса Box имеют два отличия от объектов класса athlete:

- С ними не ассоциируются гимнасты.
- Объекты класса Box снабжены метками.

Информационные рамки (объекты класса Box) автоматически располагаются одна под другой в правой части экрана.

Например, чтобы вывести на экран время, вы должны следующим образом объявить объект:

```
Box display ("Time:");
```

Теперь в любой момент вы можете использовать эту рамку для вывода на экран значения времени, которое «хранится» в уже созданных вами часах (объекте mywatch):

```
display.say(mywatch.time());
```

В данном случае в этих двух объявлениях указано, что слово Time: является меткой рамки, внутри которой отображается значение времени.

Если вы объявляете более одного объекта класса Box, то последующие объекты располагаются один под другим. На уроке 14 вы научитесь перемещать эти рамки (объекты типа Box).

Первый этап великого похода

С помощью объектов класса Robot мы воспроизведем ситуацию, в которой вы и ваш учитель, оказавшись в лабиринте, исследуете вероятные пути выхода.

Объекты типа Robot

Объекты класса Robot представляют собой робота внутри комнаты, через стены которой робот либо может, либо не может пройти. Стандартные объекты класса Robot располагаются в пустой комнате в левом верхнем углу экрана. При своем первом появлении робот повернут в определенном направлении, а именно на восток. Подобная ситуация показана на рис. 2.1.

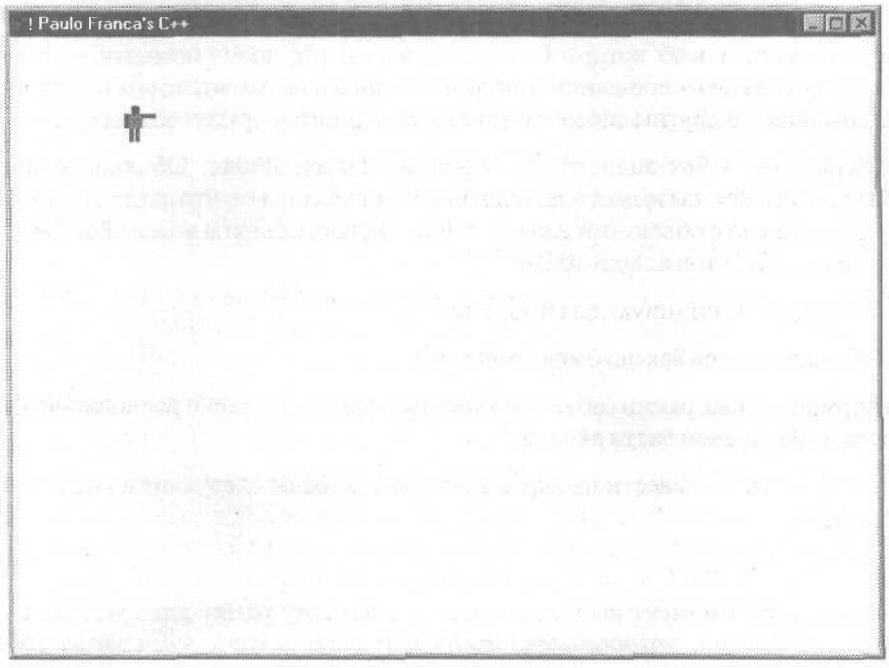


Рис. 2.1. Объект типа Robot

Чтобы использовать в программе объект класса Robot, необходимо объявить один (и только один) такой объект. Во всех примерах этой книги объявляется один объект класса Robot с именем Tracer.

Картинка, показанная на рис. 2.1, была создана с помощью приведенной ниже программы clrobot1.cpp.

```
#include "franca.h"
//c1robot1.cpp
Robot Tracer;
void mainprog()
{
   Tracer.face(3);
   Tracer.step();
}
```

Возможности робота

Объект Tracer, как и любой объект типа Robot, может реагировать на следующие сообщения:

Step() — шаг вперед. При желании внутри скобок можно указать число шагов. При этом робот не контролирует наличие или отсутствие стены. Каждый шаг занимает стандартный интервал времени, так называемый timescale (см. ниже), который по умолчанию принимается равным одной секунде.

left() — поворот на 90 градусов влево.

right() — поворот на 90 градусов вправо.

seewall() — контроль стены впереди себя (на следующей позиции). При наличии стены в ответ на это сообщение выдается единица, в противном случае выдается ноль. (Это сообщение нам понадобится только на уроке 8.)

seenowall() —контроль стены впереди себя (на следующей позиции). При отсутствии стены в ответ на это сообщение выдается единица, в противном случае выдается ноль. (Это сообщение нам понадобится только на уроке 8.)

face(целое) — поворот робота в определенном направлении. Направление задается целым числом (0, 1, 2 или 3), которое соответствует направлению на север, запад, юг или восток. При задании любого другого числа используется остаток от его деления на 4.

mark() — заполнение следующей позиции заданным цветом. Если в скобках не указано число, определяющее соответствующий цвет, то по умолчанию предполагается двойка (зеленый цвет). Соответствие цветов и чисел представлено в табл. 2.1.

say("сообщение") — вывод на экран «сообщения» от имени робота. Сообщение может быть либо целым числом (в этом случае кавычки опускаются), либо состоять из нескольких слов (заключенных в кавычки). Количество символов в одной строке не должно превышать 12.

Таблица 2.1. *Соответствие цветов и чисел в сообщении mark()*

Цвет	Число
Белый	0
Красный	1
Зеленый	2
Синий	3
Голубой	4
Розовый	5
Желтый	6
Черный	7

timescale(значение) — изменение значения стандартного интервала времени. Вам этот интервал не понадобится, пока вы, например, не решите, чтобы по умолчанию каждый этап занимал 0,1 секунды. Например, если вам хочется, чтобы на каждый шаг робот затрачивал 0,5 секунды, то вы можете добиться этого следующей инструкцией:

```
Tracer.timescale(0.5);
```

Синтаксис языка C++ одинаков для управления любыми объектами: гимнастами, часами или роботами. Для отправки сообщения объекту последовательно указываются имя объекта, точка, имя сообщения и далее в скобках аргумент сообщения. Поэтому, как это показано в программе c1robot1.cpp, чтобы объект Tracer сделал шаг вперед, используется следующая инструкция:

```
Tracer.step();
```

Когда объект Tracer появляется на экране, его рука указывает направление движения. В следующих примерах объект Tracer будет двигаться по пустой комнате, так что вам не придется беспокоиться о том, что он упрется в стену. Позднее вы узнаете, как обнаруживать на пути препятствия и избегать столкновений.

Перемещение робота

Теперь модифицируем программу clrobotl.cpp и поэкспериментируем с ней. Для начала заставим робота сделать четыре шага. Этого можно добиться повторением следующих инструкций:

```
Tracer, stepO;
Tracer.step();
```

```
Tracer.step();
Tracer.step();
```

Либо указав внутри скобок требуемое число шагов:

```
Tracer.step(4);
```

Пока этого достаточно. Впоследствии вы сможете заставить компьютер повторить ту или иную последовательность инструкций.

Разметка трассы

Чтобы контролировать путь робота, будем закрашивать ячейки, по которым он движется. Робот может закрашивать в заданный цвет только ту ячейку, которая находится прямо перед ним.

Если не принимать во внимание первую ячейку, то программа будет иметь следующий вид:

```
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();
```

С помощью этой программы объект Tracer закрашивает четыре ячейки. Начальная ячейка остается незакрашенной.

Повороты

Робот может поворачивать налево или направо, а также в любом из четырех направлений. Эти направления представлены целыми числами, как показано в табл. 2.2.

Пусть после того, как робот пройдет и закрасит четыре ячейки, нам захочется вернуть его в исходную позицию.

Как это сделать? Очень просто. Нужно заставить робота развернуться и снова сделать четыре шага. А как заставить его развернуться? Либо дважды повернуть его налево или направо (сообщение left() или right()), либо сразу развернуть в заданном направлении (с помощью сообщения face()). Если считать, что робот шел на восток, с помощью следующей инструкции развернем его на запад:

```
Tracer.face(1);
```

Таблица 2.2. Соответствие направлений и чисел в сообщении face()

Направление	Число
Север (верхняя часть экрана)	0
Запад (левая часть экрана)	1
Юг (нижняя часть экрана)	2
Восток (правая часть экрана)	3

Диалог с роботом

В любой момент робот может проинформировать вас о том, что происходит в программе. Как и гимнасты, робот умеет говорить. Например:

```
Tracer.say("Done!");
```

Кстати, вам не показалось, что робот и гимнасты внешне похожи? Может они как-то связаны между собой? Кто знает...

Теперь вы знаете как модифицировать свою программу, чтобы робот сказал, например, «going» (вперед) и закрасил четыре клеточки. А когда робот вернется назад, его можно заставить сказать «coming back» (возвращаюсь) и вернуть ячейкам первоначальный цвет.

Лабиринт

Представьте, что вы, ваш учитель и робот заблудились в лабиринте из прямоугольных ячеек. Нужно исследовать лабиринт. Для начала нужно научить робота обходить квадрат 3 на 3. Как вы понимаете, для этого робот должен сделать три шага, повернуть направо, сделать еще три шага и т. д.

Соответствующий алгоритм выглядит следующим образом:

Шаг
Шаг
Шаг
Поворот направо
Шаг
Шаг
Шаг
Поворот направо
Шаг

Шаг

Шаг

Поворот направо

Шаг

Шаг

Шаг

Поворот направо

Теперь перепишем алгоритм в программу:

```
#include "franca.h"  //c1robot2.cpp
Robot Tracer;
void mainprog()
{
Tracer.face(3);
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();        // Одна сторона готова
Tracer, right();
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();        // Вторая сторона готова
Tracer.right();
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();        // Третья сторона готова
Tracer.right();
Tracer.markO;
Tracer.step();
```

продолжение

```
Tracer.mark();
Tracer.step();
Tracer.mark();
Tracer.step();      // Последняя сторона готова
Tracer.right();
}
```

Вы, вероятно, утомились, записывая все эти шаги. Подождите до следующего упражнения! Результат выполнения программы показан на рис. 2.2.



Рис. 2.2. *Путешествие робота по квадрату*

Что неправильно в этой программе? Почему сторона квадрата состоит из 4, а не из 3 ячеек? Выходит, наша программа ошибочна?! Просто в этой программе робот, делая по 3 шага в каждом направлении, проходит квадрат со стороной, равной 4 ячейкам. Чтобы сторона квадрата равнялась 3 ячейкам, в каждом направлении нужно сделать всего два шага.

Что нового мы узнали?

В этом уроке мы научились

- Контролировать время, используя объект типа Clock.
- Выводить данные, используя объект типа Box.
- Работать с объектом типа Robot.



УРОК 3 Решение проблем

- Решение простых проблем
- Правила и синтаксис

Цель этого урока — научить вас самостоятельно работать над решением возникающих проблем. С помощью этой книги вы сможете продвинуться в этом направлении, если, конечно, будете продолжать заниматься программированием. Правда, я не могу научить вас находить решения любых проблем. Единственное, что в моих силах, — это дать вам несколько советов, как делать это самому.

В конце этого урока вы найдете формальные правила написания программ. Я специально опустил большую часть правил, чтобы вы могли сконцентрироваться на базовых концепциях: назначении той или иной инструкции, ее использовании и т. д. Однако для компилятора правила чрезвычайно важны, поэтому их краткий обзор необходим уже сейчас.

Локализация проблемы

Сложнее всего начинать. Напишите любое пришедшее вам в голову решение. Если оно окажется неправильным, не расстраивайтесь. Когда решение написано, его гораздо легче исправить, чем придумывать новое. Если решение не совсем полное, продолжайте с ним работать, делая его все более и более подробным.

Пишите все что угодно

Никогда не сидите в задумчивости над чистым листом бумаги (или пустым экраном монитора). Так вы точно ничего не добьетесь.

Изложив на бумаге первый пришедший в голову вариант решения проблемы, несколько раз перечитайте его. Правильно ли оно? Достаточно ли оно подробно? Например, следующая фраза сама по себе правильна:

Нарисовать квадрат со стороной 3

Однако она недостаточно подробна и для компьютера не подходит. Вы должны объяснить ему, что вы понимаете под рисованием квадрата со стороной 3!

Дробление проблемы

До тех пор пока компьютер не будет в состоянии понять, как выполнить каждый свой шаг, необходимо постоянно уточнять путь решения проблемы. Основная идея в том, чтобы выделить более мелкие задачи, решение которых вы знаете. Этим вы приближаете общее решение. Постарайтесь разбить задачу не просто на элементарные части, а по возможности на те, решение которых вам известно. Например, задачу рисования квадрата можно свести к поэтапному рисованию отдельных линий:

Рисование линии
Поворот направо
Рисование линии
Поворот направо
Рисование линии
Поворот направо
Рисование линии
Поворот направо

Было бы удивительно, если бы компьютер уже знал, как рисовать линию. Но не беда. Вы можете объяснить ему это следующим образом:

Закрашивание ячейки
Шаг вперед
Закрашивание ячейки
Шаг вперед

Это описание дает нам искомый алгоритм.

Пользуйтесь готовыми решениями

Наконец, позволю себе дать еще один важный совет. Разбивая проблему на части, постарайтесь свести ее к готовым решениям. Что означает свести к готовым решениям? Это означает, что решения простых задач, полученные вами ранее, станут составными частями решения проблемы в целом. Чем больше у вас будет готовых решений, тем больше времени вы сэкономите. Тем не менее, как быть, если у вас нет под рукой подходящего фрагмента кода?

В этом случае вам придется составлять его самому. Но при этом постарайтесь сделать это так, чтобы вашу программу впоследствии можно было использовать при решении других задач. Пытайтесь строить свое программное обеспечение или, точнее, свои функции (что это значит, вы узнаете на следующем уроке) таким образом, чтобы без каких-либо изменений их можно было использовать повторно.

Сообщения об ошибках

Почему компьютер так легко поставить в тупик? В этом упражнении представлены некоторые наиболее распространенные программные ошибки. Я хочу, чтобы вы разобрались с основными типами сообщений об ошибках. Заметим однако, что во многих случаях сообщения об ошибках не слишком точно объясняют возникшую проблему.

Возьмем программу c_sal.cpp и последовательно внесем в нее несколько изменений:

- Уберем точку с запятой в конце инструкции Sal.ready().
- Уберем одну из скобок в инструкции Sal.ready);.
- Уберем обе скобки в той же инструкции.
- Уберем строку tfindude.
- Уберем строку Sal.say("Hi!");.

Скорее всего, в каждом из этих случаев вы получите сообщение об ошибке, показывающее, что компьютер не знает, что вы от него хотите. Постарайтесь понять, почему компьютер не понял те или иные команды. Например, убирая точку с запятой, мы заставляем компьютер считать, что соответствующая инструкция не заканчивается, а переходит на следующую строку, но, поскольку следующая инструкция для компьютера ничего не означает, естественно компилятор не знает, как на нее реагировать:

```
Sal.ready() Sal.say("Hi!");
```

Это может показаться легко исправимой мелкой ошибкой, но компьютер, работающий ло строгим правилам, не способен понять, что именно неправильно.

Большинство разобранных выше примеров относится к категории *синтаксических ошибок*, состоящих в нарушении простых правил составления инструкций. Например, на каждую открывающую скобку должна приходиться одна закрывающая; все имена объектов должны быть объявлены и т. д.

Однако последняя ошибка в списке не приведет к сообщению об ошибке. Ваша программа даже заработает, но она будет неправильной, поскольку слова «Hi!» под картинкой не появится. В этом случае компилятор ошибки не обнаружит. Правила синтаксиса соблюдены. Тем не менее программа неправильна, поскольку делает не то, что вы от нее ждете.

Запомните, компьютер всегда делает только то, что ему приказано. И если что-то идет не так, то, вероятнее всего, он не получил четких инструкций. Очевидно, что, поскольку компьютеры обладают упрощенным «мышлением», соответствующие правила должны строго соблюдаться. Малейшая ошибка может привести к совершенно неожиданному результату.

Правила и соглашения

До сих пор я довольно поверхностно объяснил правила построения программ. Главным образом потому, что мне хотелось скорее довести до вас назначение основных компонентов программы, чем приводить конкретные правила их использования. Тем не менее важно знать эти правила, и я приведу их в этом разделе. Возможно, теперь, когда вы уже знакомы с назначением некоторых программных конструкций, вам будет легче понять правила их написания.

Идентификаторы

Для распознавания (идентификации) объектов, используемых в программе, каждому объекту назначается свой *идентификатор,* являющийся, по сути, именем объекта. Вы можете выбрать идентификатор по своему усмотрению, исходя из следующих правил:

- Идентификатор должен начинаться с буквы или символа подчеркивания (_).
- В идентификатор могут входить буквы алфавита, цифры (от 0 до Э) и символы подчеркивания.
- В идентификаторе не может быть более 32 знаков (можете, конечно, написать идентификатор хоть из миллиона, но компьютер прочтет только первые 32).
- В идентификаторе нельзя использовать пробел.
- В качестве идентификаторов нельзя использовать ключевые слова (что это такое, вы узнаете позже).

Советую давать своим объектам такие имена, которые бы напоминали о назначение этих объектов.



Длинные имена через некоторое время могут вам надоесть.

Ниже представлены примеры неправильных идентификаторов:

- 2waystop (начинается с цифры).
- my number (содержит пробел).
- this-number (знак минус не разрешен).
- "Sal" (кавычки не разрешены).

А вот примеры правильных идентификаторов:

- twowaystop
- mynumber
- thisnumber
- Sal

Ключевыеслова

Каждое ключевое слово несет компилятору специальную информацию. На некоторые ключевые слова, например слово void, вы, наверное, уже обратили внимание. Ключевые слова нельзя использовать в качестве имен объектов. По мере изучения этой книги, вы узнаете еще много ключевых слов. В большинстве компиляторов ключевые слова выделяются (цветом или начертанием).

Типы и классы

Типы и классы означают одно и то же (по крайней мере на данном этапе), и будем считать эти термины взаимозаменяемыми. Соответствующие технические различия между ними объясняются на следующих уроках.

Типы или классы обозначают общие характеристики целых групп объектов. Скоро вы познакомитесь поближе с типом данных, представленных целыми числами и состоящих только из них. Тип int (целые) состоит из чисел, у которых отсутствует дробная часть. Над объектами типа int можно выполнять различные арифметические действия. Вы уже научились обращаться с данными других типов, например с гимнастами (данными типа athlete). Их можно заставить выполнять разного рода команды, они также умеют «говорить». Все данные одного типа должны иметь сходные характеристики. Так, например, все объекты типа athlete умеют выполнять указанные выше команды.

Необходимо сообщать компилятору класс каждого используемого вами объекта, чтобы компилятор мог проверять корректность ваших действий над своими объектами.

Вы также можете создавать собственные классы. Например, athlete — это класс, который я специально создал для вас. На следующих уроках вы узнаете, как можно создавать новые классы либо из существующих классов (например, класса athlete), либо начиная с нуля.

Объявления

Объявление (declaration) объекта состоит из имени класса, за которым следует имя объекта. Соответствующий синтаксис имеет следующий вид:

```
имя_класса имя_объекта;
```

Идентификатор имя_класса должен быть именем вполне конкретного класса, существующего в вашей программе (например, классы athlete и clock объявлены в заголовочном файле franca.h), в то же время идентификатор имя_объекта вы назначаете сами. Можно объявлять более одного объекта, разделяя их идентификаторы запятыми, как это сделано, например, в следующей инструкции:

```
athlete Sal, Sally;
```

Идентификатором можно пользоваться только после того, как он объявлен. Объявлять объекты и переменные можно в любом месте программы. Например;

```
Clock mywatch;          // Объявление объекта mywatch типа Clock
int number_of_times; // Объявление целой переменной number_of_times
```

Сообщения

Объект можно заставить выполнять определенные действия. Для этого программа посылает объекту *«сообщение»*. В свою очередь объекты могут реагировать только на те сообщения, для обслуживания которых они были разработаны.

Для посылки сообщения объекту необходимо последовательно написать имя объекта (идентификатор), точку и имя сообщения, за которым следуют скобки. В скобках при необходимости указывается аргумент сообщения. Соответствующий синтаксис представлен ниже:

```
имя_объекта.сообщение();
```

В следующем примере объекту Sal посылается сообщение up():

```
Sal.up();
```

А теперь объекту Sal посылается сообщение up() с аргументом 2:

```
Sal.up(2);
```

Аргумент показывает, что следующая за данной программная инструкция должна быть выполнена только через 2 секунды.

На техническом языке в C++ вместо термина *сообщение* используется термин *функция-член*. *С* функциями-членами мы познакомимся на уроке 16.

Комментарии

Комментарии включаются в программу, чтобы вам или кому-то еще было проще ее понимать. Компьютер на комментарии не реагирует. В программах на C++ есть два способа делать комментарии:

- Размещать комментарий после двойной косой черты (//).
- Размещать комментарий между открывающей (/*) и закрывающей (*/) парой символов.

В последнем случае комментарий может быть многострочным:

```
/* Этот комментарий
занимает целых
три строки */
```

Такой комментарий можно даже вставлять непосредственно в инструкцию:

```
int/* это комментарий */ iapcount;
```

Самостоятельная практика

Рассмофим несколько примеров,

Поиск ошибки

Следующая профамма — это испорченная версия нашей первой профаммы c_sal.cpp:

```
include "franca.h"
athlete Sal;
void mainprog()
{
sal.ready();
Saf.say(Hi);
}
```

Здесь имеются три ошибки. Сравните ее с оригиналом и найдите ошибки. Затем попытайтесь выполнить неправильную программу и изучите сообщение об ошибках.

Написание программ

Сначала напишите программу, изображающую на экране трех гимнастов и их имена. Объявите трех гимнастов (их имена выберите сами) и пошлите сообщение каждому, чтобы он занял положение готовности и «назвал» свое имя.

Затем напишите программу, изображающую на экране картинку, показанную на рис. 3.1 (Сэл и его двойники). Теперь нужно объявить четырех гимнастов и послать каждому свое сообщение.

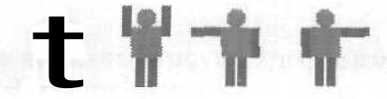


Рис. 3.1. *Сэл и его двойники — ваша первая самостоятельная программа*

Поиск ошибок

Следующая программа представляет собой еще одну испорченную версию программы c_sal.cpp:

```
#include "franca.h"
void mainprog()
```

```
{
ready;
Sal.say("Hi!");
}
```

Здесь также имеются три ошибки. Найдите их и, выполнив программу, изучите сообщения об ошибках.

Правильные идентификаторы

Среди представленных ниже идентификаторов укажите правильные:

```
2ndTime
apollo-13
apollo13
BigTime
my friend
my.friend
my_friend
void
```

Рисование

Напишите программу, с помощью которой ваш робот мог бы создать картинку, аналогичную показанной на рис. 3.2.

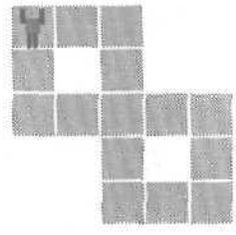


Рис. ***3.2.*** *Робот-художник*

Нарисовать эту картинку можно двумя способами:

- Нарисовать квадрат, перейти в правый нижний угол этого квадрата и затем нарисовать другой квадрат.
- Нарисовать все линии непрерывным перемещением робота.

Пусть каждый шаг занимает одну секунду. Можете ли вы рассчитать, какое решение будет быстрее? Учитывая, что робот знает, как рисовать квадрат, ответьте, какое решение проще?

Поиск идентификаторов

Изучите следующую программу. В каких идентификаторах есть ошибки? А какие вообще не объявлены?

```
void mainprog()
{
   athlete bill, john doe, ann3;
   Clock 3 days, my-clock, O'Hara, other;
   Box show;
   show.say(thetime);
   bill.say(other.time());
   Marie.ready();
   Robert.up();
}
```

Что нового мы узнали?

В этом уроке мы научились

0 Находить решения простых проблем.

☑ Понимать синтаксис и правила.

Часть II

Функции и выражения

На уроках части II вы научитесь разрабатывать функции и простые арифметические выражения. Вы узнаете, как использовать функции и их аргументы, а также как оперировать со значениями функций и возвращать результат. Функции позволят вам усовершенствовать свою технику программирования.

В конце части II вы найдете формальное обсуждение правил использования функций и арифметических выражений.



УРОК 4 Функции

- Создание функций
- Расширение возможностей функций с помощью аргументов
- Создание заголовочных файлов
- Понятие области видимости

На этом уроке с помощью нашего неутомимого гимнаста Сэла мы научимся создавать простейшие функции. Функцией можно сделать любое упражнение, которое умеет делать Сэл. Например, если один раз научить Сэла выполнять подскок, то учить его снова уже не придется. Все, что вам нужно будет сделать, — это создать функцию, объясняющую, как нужно выполнять подскок.

В дальнейшем вы убедитесь, насколько полезны функции. Вы также научитесь сохранять нужные функции в заголовочных файлах, чтобы их можно было использовать снова в любой создаваемой вами программе.

В конце урока будет введено понятие области видимости. Поскольку компилятор трактует каждую функцию как отдельную программную сущность, то имена, присвоенные всем ее переменным и объектам, действительны только внутри области видимости данной функции. На самом деле это очень хорошо, так как позволяет не запоминать все идентификаторы программы. Тем не менее, чтобы избежать конфликтов имен, вы должны знать несколько правил области видимости.

Понятие функции

Иногда определенное действие состоит из группы отдельных команд. В этом случае гораздо удобнее обратиться сразу ко всей группе, а не выписывать все необходимые команды по отдельности.

Рассмотрим следующую группу команд:

```
Sal.up();
Sal.left();
Sal.up();
Sal.right();
Sal.up();
Sal.ready();
```

Очевидно, что все эти действия определяют одно упражнение. Если мы заставим компьютер отождествить эту группу действий с одним именем, например *влево-вправо,* то, дав команду *влево-вправо,* мы на самом деле попросим Сэла выполнить некоторую последовательность действий. Другими словами, всю описанную выше последовательность инструкций мы определяем как единую функцию, которая описывает, как выполняется эта последовательность. Аналогичный элемент в других языках программирования называют процедурой или подпрограммой. Хотя в терминах C++ такие части программы называют *функцией* (function), по-прежнему вполне применим и исходный термин — *процедура* (procedure).

ПРИМЕЧАНИЕ Функция представляет собой группу инструкций, которой назначено некоторое имя. Все эти инструкции могут быть выполнены посредством вызова функции по ее имени.

Благодаря функциям компьютер становится сообразительнее. Однажды определив функцию, вам больше не понадобится повторять все входящие в функцию инструкции. Компьютер уже будет знать, как они выполняются! Кроме того, программа становится понятнее.

СОВЕТ Как и люди, компьютер не может запомнить функцию роз и навсегда. Поэтому вам необходимо включать код вашей функции в каждую программу, где вы собираетесь ее использовать.

Бы, конечно, заметили, что последовательность движений Сэла составляет единое упражнение. Инструктор по гимнастике может дать отдельное имя и, например, такой последовательности движений:

Позиция готовности

Прыжок

Позиция готовности

Инструкторы называют эту последовательность действий *подскоком*. Таким образом, когда инструктор хочет заставить группу выполнить подскок, то уже не объясняет всю последовательность действий.

Например, однажды показав, как выполняются упражнения *подскок* и *влево-вправо*, инструктор будет отдавать свои команды следующим образом:

Подскок

Влево-вправо

Подскок

Обратите внимание, насколько упростились команды. Вместо того чтобы давать подробное описание всей последовательности движений, инструктор теперь просто называет имя нужного упражнения.

Ту же стратегию можно применить и на вашем компьютере — можно сгруппировать несколько инструкций и присвоить им единое имя. Теперь, чтобы выполнить функцию, вам достаточно просто указать ее имя, а компьютер выполнит каждую входящую в нее инструкцию.

ПРИМЕЧАНИЕ Робот тоже может научиться работать с функциями. Например, можно создать функцию рисования линии заданного размера.

Создание функции

Попробуем создать функцию. Поскольку прежде, чем писать любую программу (в том числе функцию), нужно точно понять, что мы хотим от нее получить, для начала сделаем описание программы.

Функция подскока:

Команда Сэлу подпрыгнуть

Команда Сэлу занять исходное положение

Окончание функции подскока.

На языке C++ это будет выглядеть следующим образом;

```
void JumpJack()
{
   Sal.up();
   Sal.ready();
}
```

Имя функции

Компьютеру нужно объяснить, во-первых, действие функции, а во-вторых, ее имя. По-русски это звучало бы так: *это функция выполнения подскока*. Эта фраза объясняет, что функция выполнения подскока состоит из некоторой последовательности инструкций.

Границы функции

Где первая и где последняя инструкция в принадлежащей функции последовательности инструкций?

Компьютеру нужен точный ответ на этот вопрос. Чтобы выделить первую и последнюю инструкции, вокруг интересующей нас последовательности действий ставятся открывающая и закрывающая фигурные скобки {}. Функция начинается с открывающей фигурной скобки и заканчивается закрывающей. Внутри этих скобок инструкции обычно сдвигают вправо. Для компьютера это неважно, но для нас это очень полезно и является хорошим стилем программирования.



ВНИМАНИЕ С технической точки зрения вся функция рассматривается как единая составная инструкция (что это такое, вы узнаете позже). Но если в конце строки с именем функции вы ставите точку с запятой, то тем самым даете понять компилятору, что ваша инструкция на этом заканчивается (еще до начала тела функции).

В программе перед именем функции ставится слово void, о назначении которого будет рассказано ниже, а за именем функции следуют круглые скобки {). В конце строки с именем функции точка с запятой не ставится.

Рассмотрим программу c2jmpjck.cpp:

```
//            c2jmpjck.cpp
// Составлена Paulo Franca
// Последняя доработка 05.01.96
// В этой программе для выполнения упражнений Сэл использует
// функцию JumpJack
#include "franca.h"    // Вставка программ
                       // из программного обеспечения книги
athlete Sal;           // Создание объекта Sal класса athlete
void JumpJack()        // Начало функции JumpJack
{
   Sal.up();           // Прыжок
   Sal.ready();        // Исходное положение
}                      // Конец функции JumpJack
void mainprog()        // Начало основной программы
{
   Sal.ready();        // Исходное положение
   JumpJack();         // Подскок!
}                      // Конец основной программы
```

В программу входят коды функций JumpJack() и mainprog(). Заметим, что mainprog — это имя функции, в которой вы объясняете компьютеру, что на самом деле вы от него хотите. Эта функция в принципе аналогична всем остальным, за исключением того, что должна присутствовать в каждой вашей программе. В функции mainprog() можно вызывать другие функции. В нашем примере это была функция JumpJackO.



ПРИМЕЧАНИЕ Функция должна быть описана в программе до ее первого использования.

Если мы хотим, чтобы Сэл выполнил еще несколько подскоков, нужно просто включить в программу две дополнительные инструкции. Тогда функция mainprogO будет выглядеть следующим образом:

```
void mainprogO    // Начало основной программы действий
{
   Sal.ready();   // Исходное положение
   JumpJackO;     // Подскок!
```

```
    JumpJack();     // Еще подскок!
    JumpJack();     // И еще подскок!
}                   // Конец основной программы действий
```

Запомните, функция — это, по сути, набор действий, которому вы присвоили собственное имя. Благодаря функциям ваши программы становятся понятнее, поскольку отпадает необходимость снова и снова описывать всю последовательность шагов. Кроме того, перечисленные в функции действия могут выполнять самые разные объекты.

Управление несколькими **объектами**

Предположим, что у Сэла есть подружка Салли, тоже занимающаяся гимнастикой. Включив в программу следующее объявление, мы легко получаем сразу двух гимнастов:

```
athlete Sal, Sally;
```

Это объявление сообщает компьютеру, что теперь имеются два объекта типа athlete и, следовательно, мы можем вывести на экран оба этих объекта;

```
Sal.ready();
Sally.ready();
```

Как заставить Сэла выполнить подскок, мы уже знаем. А что, если нам захочется, чтобы то же самое сделала Салли? В этом случае бесполезно вызывать функцию JumpJack(), поскольку все перечисленные в ней упражнения предназначены для Сэла. Что же делать? Есть два простых решения:

- Можно специально для Салли создать еще одну функцию JumpJack().
- Можно просто перечислить для Салли все действия, входящие в подскок.

Оба эти решения некорректны. Мы ведь уже описывали, как выполняется подскок, и это описание одинаково для Сэла, Салли и для любого другого гимнаста. (В конце концов, опытные инструкторы не объясняют *каждому* члену группы, как выполняется то или иное упражнение, не правда ли?)

Аргументы функций

Было бы здорово переделать функцию так, чтобы она описывала выполнение подскока *любому* гимнасту. Это должно быть описание *способа выполнения упражнения,* который не зависит от того, *кто* именно его выполняет.

В каком-то смысле это напоминает театральную постановку. Автор пишет пьесу, в которую входят те или иные персонажи. При этом он, как правило, не знает, кто конкретно будет играть их роли. Одну и ту же роль могут играть разные актеры, но актер и роль не привязаны друг к другу раз и навсегда, и каждый актер играет за свою жизнь множество разных ролей.

Аргументы и параметры

Когда мы создаем функцию, то обычно не знаем заранее, какие объекты будут ее аргументами. Следовательно, мы говорим о фиктивных или подразумеваемых объектах, которые называют *параметрами* (parameters). При вызове функции параметр заменяется реальным объектом. Реальные объекты, которые при выполнении функции используются вместо параметров, называют *аргументами* (arguments). Все прямо как в театре.

Итак, мы хотим, чтобы функция JumpJack() программы c2jmpjck.cpp могла управлять действиями как Сэла, так и Салли (или любого другого гимнаста, который может появиться в дальнейшем). Этого можно добиться, заставив функцию работать не с конкретным, а с любым объектом типа athlete.

1. Объясните компьютеру, как заставить фиктивного гимнаста выполнить упражнение. Вы можете дать этому гимнасту какое-то имя, например *некто* (параметр somebody). Для этого *некто* включите в функцию все знакомые нам инструкции выполнения упражнения. Но у нас нет гимнаста с именем *некто*, а есть только Сэл и Салли. Следовательно, *некто* — это и есть параметр.
2. Научив гимнаста *некто* выполнять подскок, заставьте Салли (или Сэла) сыграть роль этого *некто* в вашей функции. Салли — это объект, реально существующий в вашей программе (так же, как актеры существуют в реальной жизни), и он может занять место фиктивного *некто*.

Посмотрим, как это сработает в программе c2jmpbdy.cpp:

```
//              c2jmpbdy.cpp
//
#include "franca.h"
athlete Sal, Sally;
void JmpJack(athlete somebody)
{
   somebody.up();
   somebody. ready();
}
void mainprog()
{
   Sal.ready();
   JmpJack(Sal);
   Sally ready();
   JmpJack(Sally);
}
```

В функции JmpJack() мы использовали объект sombody, которого, как мы знаем, на самом деле не существует. Тем не менее мы объяснили ему, какие действия надо

выполнить, чтобы совершить подскок. Если вы посмотрите на основную часть программы, то заметите, что функция JumpJack() упоминалась в ней дважды:

```
JmpJack(Sal);
JmpJack(Sally);
```

В первом случае внутри фигурных скобок находится Sal. Что это означает?

Мы выполняем функцию JumpJack(), но вместо фиктивного параметра somebody в качестве аргумента указываем реальный объект Sal. Все движения будут выполнены реальным гимнастом Сэлом, а не каким-то гипотетическим *некто.* Во втором случае в качестве аргумента указан объект Sally. После выполнения программы экран вашего компьютера будет выглядеть так, как показано на рис. 4.1.

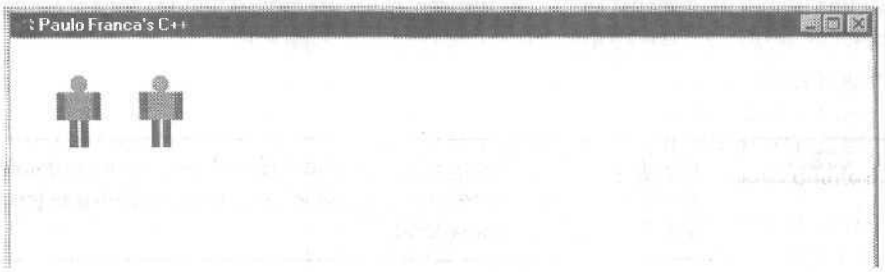


Рис. 4.1. Результат выполнения программы c2jmpbdy.cpp

Используя новый метод попытаемся поменять местами аргументы, поочередно объявляя Сэла и Салли:

- Пусть гимнасты займут исходные позиции и объявят свои имена. Затем вызовем функцию JmpJack() с каждым гимнастом в качестве аргумента. Посмотрите на результат.
  - Для этого нужно, чтобы перед вызовом функции JmpJack() Сэл сказал «Сэл», а Салли сказала «Салли».
- Теперь вызовем функцию JmpJack() в обратном порядке: сначала с Салли, а затем с Сэлом. Посмотрите на результат.
  - Другими словами, сначала записываем инструкцию JmpJack(Sally);, а затем — инструкцию JmpJack(Sal);.

Аргументы по умолчанию

Иногда бывает удобно определить для параметров значения по умолчанию. Особенно это полезно, когда в качестве аргумента чаще других приходится использовать один и тот же объект (аргумент по умолчанию).

Пусть, например, мы хотим, чтобы в большинстве случаев упражнение выполнял Сэл и только иногда Салли. В определении функции мы можем задать аргумент по умолчанию, присвоив параметру нужное значение. Например, если мы следую-

щим образом изменим заголовок функции JmpJack() то, если при вызове функции никого другого не задано, Сэл будет выступать в роли гимнаста по умолчанию («штатного» гимнаста):

```
void JmpJack(athlete sombody=Sal)
```

Таким образом, инструкция

```
JmpJack();
```

приведет к тому же результату, что и инструкция

```
JmpJack(Sal);
```

Это дает нам возможность при вызове функции просто опускать наиболее часто встречающееся значение ее аргумента. В случае необходимости мы можем включить туда нужный нам аргумент. Например, если мы хотим использовать Салли, то можем сделать это, вызвав функцию следующим образом;

```
JmpJack(Sally);
```



СОВЕТ Когда вы определяете аргумент по умолчанию, то последний должен быть предварительно определен и известен функции (см. раздел об области видимости).

Функции нескольких аргументов

Можно создавать функции с несколькими аргументами. Например, мы можем описать функцию выполнения подскока, в качестве аргументов которой указаны оба гимнаста. В C++ в одной программе возможно сосуществование нескольких функций с одним именем, но с различным числом или типом (классом) аргументов.

Давайте создадим новую функцию JmpJack(), описывающую выполнение упражнения обоими гимнастами. Очевидно, что для этого можно просто вызвать нашу прежнюю функцию JmpJack() для каждого из гимнастов:

```
void JmpJack(athlete first, athlete second)
{
    JmpJack(fikst);
    JmpJack(second);
}
```

Однако так вы заставите гимнастов упражняться по очереди. Можно, конечно, и так, но как заставить их работать одновременно? Фактически полностью одновременное выполнение упражнений невозможно, поскольку компьютер выполняет инструкции последовательно. Мы попытаемся контролировать время, в течение которого гимнаст находится в каждой позиции.

Чтобы создать иллюзию одновременного выполнения упражнений, мы можем заставить первого гимнаста занять исходное положение за 0 секунд, а затем заставить второго сделать то же самое за одну секунду. Аналогично поступим с другими

движениями. Сейчас большинство компьютеров выполняет операции так быстро, что вы вряд ли их заметите. Но картинка, которая остается на экране достаточно продолжительное время (одну секунду), четко отпечатается в вашем сознании.

Ниже представлен алгоритм работы функции.

Функция JmpJack() для выполнения одновременного подскока двух гимнастов:

Первый гимнаст остается в *исходном положении* ноль секунд.

Второй гимнаст остается в *исходном положении* одну секунду.

Первый гимнаст находится в *прыжке* ноль секунд.

Второй гимнаст находится в *прыжке* одну секунду.

Первый гимнаст остается в *исходном положении* ноль секунд.

Второй гимнаст остается в *исходном положении* одну секунду.

Сможете ли вы самостоятельно написать функцию и оценить результат? Начните со следующей инструкции:

```
void JmpJack(athlete first, athlete second)
```

Значения и ссылки

Обычно когда вы вызываете функцию с аргументом, то функция использует копию этого аргумента. Другими словами, когда выполняется следующая инструкция, компьютер создает копию объекта Sally и затем использует ее в функции JmpJack():

```
JmpJack(Sally);
```

С самим же объектом Sally ничего не происходит, вне зависимости от того, какие операции выполняются в функции. В C++ при вызове функции это обычное явление. Вы можете быть уверены, что функция никак не повлияет на исходный объект.

В рассмотренном нами примере разница между оригиналом и копией незаметна. Вместо исходного гимнаста создается другой, который действует согласно записанным в функции инструкциям. Глядя на экран, вы не сможете сказать, кто выполняет упражнения — истинный гимнаст или его копия.

Увидеть, к чему приводит использование копий, можно с помощью объекта типа Clock (часы). В следующем примере мы сначала создадим объект типа Clock, который назовем timer (таймер), а затем вызовем функцию перезапуска таймера. Вместо того чтобы перезапустить оригинальные часы, функция перезапустит их копию.

Наблюдать это можно с помощью информационных рамок, показывающих время до начала, в процессе и после выполнения функции. Поэтому создадим три рамки и пометим их соответственно Before:, During: и After:. Проблему иллюстрирует программа c2clkcpy.cpp:

```
#include "franca.h"                    // c2clkcpy.cpp
// Эта программа демонстрирует использование в функциях копий аргументов
Box before ("Before:"), during ("During:"), after ("After:");
void zero(Clock clone)
{
   clone.reset();                  // Перезапуск часов
   during.say(clone.time());       // Вывод времени
}
void mainprog()
{
   Clock timer;
   timer.wait(1);                  // Пауза в 1 секунду
   before.say(timer.time());       // Вывод времени
   zero(timer);                    // Вызов функции перезапуска
   after.say(timer.time0);         // Вывод времени
}
```

Если вы запустите эту программу, то увидите на экране три числа. Первое число показывает, сколько секунд прошло с момента запуска программы. Из-за сообщения wait(1) оно должно равняться примерно 1.

Второе число показывает, сколько секунд прошло после того, как часы были перезапущены функцией zero(). Это число должно равняться 0.

Третье число показывает, сколько времени прошло с того момента, как часы были перезапущены последний раз. Вы, наверное, полагаете, что это число должно быть примерно равно нулю, поскольку, будучи аргументом функции, объект timer был обнулен. Но это не так! На самом деле число, которое вы увидите, будет равно 1 или чуточку больше. Это говорит о том, что был обнулен не объект timer, а его копия. Убедитесь в этом на рис. 4.2.

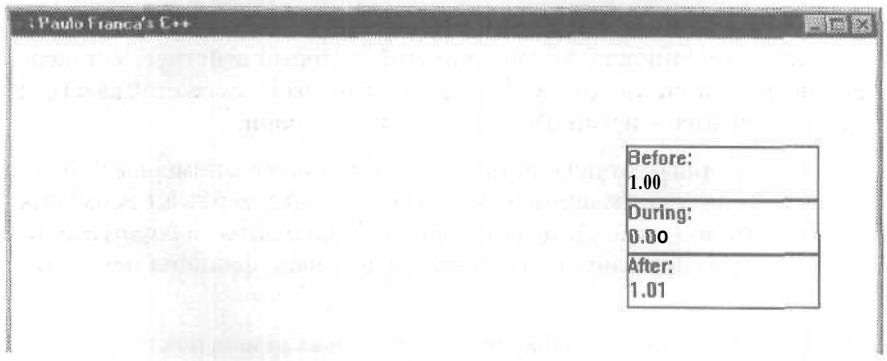


Рис. 4.2. *Результат выполнения программы c2clkcpy.cpp*

Защита ваших объектов от ваших же функций на первый взгляд кажется прекрасной идеей, но как быть, если вам нужно работать с исходным объектом.

Передача по ссылке или по значению

C++ позволяет показать, что функция предназначена для работы с оригиналом объекта, а не с его копией. В этом случае говорят, что параметр передается не по *значению* (value), а по *ссылке* (reference). Передача по ссылке выполняется очень просто. Отметьте, какие параметры вы хотите передать по ссылке (передача по значению происходит по умолчанию). В C++ для этого перед именем параметра ставится символ амперсанда &. В нашем примере измененная функция будет иметь следующий вид:

```
void zero(Clock &clone)
{
   clone.reset();           // Перезапуск часов
   during.say(clone.time()); // Вывод времени
}
```

Вот и все изменения! Все инструкции в функции остались прежними. Благодаря символу ссылки (&) можно быть уверенным, что все операции будут выполнены над исходным объектом. Теперь можете изменить программу и оценить результат.



ПРИМЕЧАНИЕ Если у вас имеется более одного параметра, то символ & нужно указать перед всеми параметрами, которые вы хотите передавать по ссылке.

Применение ссылок

Очевидно, что ссылки применяются, когда нужно изменить исходный аргумент функции. Бывают и другие ситуации, в которых удобно использовать ссылки.

На создание копии объекта компьютер затрачивает определенное время. В большинстве случаев это несущественно. Однако чем сложнее объект, тем больше времени требуется на его копирование. Например, объекты типа athlete являются достаточно сложными. Если вы уверены, что не нуждаетесь в защите вашего оригинала от модификации, а сложность вашего объекта достаточно велика, вы также можете воспользоваться передачей по ссылке.

Обучение объекта

Вы уже наверное обратили внимание, что вызов функции похож на передачу сообщения объекту. В обоих случаях выполняется некоторая последовательность действий. Когда вы посылаете сообщение объекту, вы вызываете функцию, связанную с объектами данного класса. Такие функции называются функциями-членами. Более подробно о функциях-членах вы узнаете на уроках 16 и 17.

Когда создавался класс athletes, он предназначался для обработки сообщений ready, up, Left и right. Это достигалось созданием функции-члена для каждого из

этих действий. Я мог бы включить сюда и функцию для подскока, но не захотел лишить вас удовольствия создать ее самостоятельно.

Функции-члены и функции-не-члены

У объекта имеются функции обработки всех предназначенных для него сообщений. Например, объекты класса athlets имеют следующие функции:

```
ready()
up()
left()
right()
say()
```

Эти функции являются частями объекта и не могут выполняться независимо от него. Говоря формальным языком, они представляют собой *функции-члены* класса athlets. Функции-члены могут быть определены только после определения соответствующего класса. Мы научимся это делать на следующем уроке. С другой стороны, функция JmpJack() не относится ни к одному из классов, и именно поэтому мы смогли ее создать. Функции, которые не являются частями какого бы то ни было класса, называются *функциями-не-членами*.

Заголовочные файлы

Если одну из функций вы хотели бы использовать сразу в нескольких программах, то вам придется копировать ее в каждую новую программу. Это, согласитесь, не очень удобно. Было бы лучше, если бы можно было научить компьютер самому выбирать и копировать необходимую вам функцию.

Функцию (как и любую другую программу) можно сохранить в файле. Затем, чтобы сделать себе копию функции, можно использовать предназначенную для этого директиву tfindude. Я подготовил для вас несколько функций, но если вы решите копировать их по одной, то быстро откажетесь от этого занятия. Поэтому все свои функции я разместил в заголовочном файле franca.h. Если вы используете следующую инструкцию, то просите компьютер найти файл franca.h и скопировать его в вашу программу:

```
#include "franca.h"
```

То же самое вы можете делать со своими функциями и программами.

Если функцию JumpJack() сохранить на диске, например в файле jumpjack.cpp, то вы можете скопировать эту функцию в программу с помощью следующей строки:

```
#include "jumpjack.cpp"
```

Теперь в своей программе вы не увидите функции JumpJack() (как вы не можете видеть функций из файла franca.h). Тем не менее непосредственно перед тем, как компилятор начнет транслировать вашу программу на машинный язык, копия запрашиваемой программы появится там, где находится директива #include.

Чтобы компилятор мог найти файл, который необходимо включить в программу, этот файл должен находиться в том же каталоге, что и ваша программа (например, каталог C:\franca). Если это не так, нужно указать полный путь к файлу:

```
#include "c:\franca\jumpjack.cpp"
```

Когда вы хотите включить в программу файлы, находящиеся в ваших каталогах, то имя файла заключается в двойные кавычки. В других случаях вам может понадобиться включить в программу файлы, находящиеся в каталогах компилятора (зачем это делать, вы узнаете позже). Тогда вместо двойных кавычек имя файла заключается в угловые скобки. Например:

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

Директивы

Директива (directive) — это инструкция компилятору. Она объясняет, что вы хотите сделать, перед тем как ваша программа будет откомпилирована. Директивы не относятся к языку C++, скорее их можно отнести к компилятору.

Компилятор распознает директиву по знаку фунта (#) перед первым символов строки. Имеется несколько директив компилятора, но нас будет интересовать толькодирективатfinclude.

Создание заголовочных файлов

Файлы, содержащие части программ, предназначенные для копирования в другие программы, называются *заголовочными файлами* (header file) и обозначаются расширением .h (вместо .cpp). В программу можно включать и файлы с расширением .cpp, но если хотите работать профессионально, то должны сохранять файлы с расширением .h. Это делается с помощью команды File ➤ Store As или File ➤ Save As в среде разработки приложений вашего компилятора. Наберите имя файла и после него символы .h (см. также дополнительный материал в конце этого урока).

Самостоятельная практика

- Сделайте заголовочный файл из функции JmpJack().
- Напишите программу, в которой используется функция из заголовочного файлаjumpj ack.h.

Область видимости

Если я скажу вам: «Мне нравится Наполеон», то вы, возможно, причислите меня к поклонникам великого императора Франции. Однако если в данный момент вы думаете о покупке торта, то, скорее всего, решите, что я имею в виду одноименный торт.

Что-то подобное может произойти, если не определить нашу *область видимости* (scope). Вам не понадобится каждый раз объяснять, имеете ли вы в виду императора или торт, если уверены, что употребляете слово в правильном контексте — области видимости, — продиктованным темой беседы.

Область видимости используется в C++ и других языках программирования и позволяет вводить различные объекты с одинаковыми именами при условии, что они находятся в разных областях видимости.

Если объект определен внутри функции, то он недоступен вне ее, то есть объект *локален* по отношению к функции. Если же объект определен вне всех функций, то он доступен для всех функций, определенных после него. Такой объект называется *глобальным*.

В программе c2jmpjck.cpp имеется глобальный объект Sal. Функция JmpJack() просто использовала этот глобальный объект для выполнения упражнения.

В последней версии программы c2jmpbdy.cpp имеются два глобальных объекта Sal и Sally. Функция JmpJack() имеет параметр somebody и таким образом в качестве аргументов ей доступны оба объекта. Объекты Sal и Sally по-прежнему остаются глобальными, но внутри функции mainprog() теперь можно определить локальные объекты Sal и Sally.

Хотя в глобальных объектах нет ничего плохого, все же лучше их избегать. Ниже приведена программа c2jmbd1.cpp, представляющая собой улучшенный вариант программы c2jmpbdy.cpp.

```
//              c2jmbd1.cpp
//
// В этой программе для выполнения упражнений используется
// функция JumpJack() с объектом класса athlete в качестве параметра
#include "franca.h"
void JmpJack(athlete somebody)
{
   somebody.up( );
   somebody.ready( );
}
void mainprog()
{
   athlete Sal, Sally;
   Sal.readyO;
   JmpJack(Sal);
```

```
    Sally.ready();
    JmpJack(Sally);
}
```

Обратите внимание, что теперь объекты Sal и Sally известны только внутри функции mainprog(). И это правильно, поскольку другим функциям они не нужны.

Повторение имен локальных объектов

Рассмотрим несколько измененный вариант этой программы. Единственное отличие состоит в имени параметра функции JmpJack(). Вместо параметра somebody используем параметр Sal.

В следующем примере задается область видимости объекта Sal:

```
void JmpJack(athlete Sal)
{
    Sal.up();
    Sal.ready();
};
```

А здесь задается область видимости объектов Sal и Sally:

```
void mainprog()
{
    athlete Sal, Sally;
    Sal.readyO;
    JmpJack(Sal);
    Sally.ready();
    JmpJack(Sally);
}
```

Вообще-то использовать одно и то же имя для обозначения разных объектов не следует. Но даже если вы забудете об этом, компьютер не растеряется. Внутри функции JmpJack() объект Sal будет обозначать параметр, который должен заменяться копией аргумента. Этот объект сначала играет роль объекта Sal, объявленного в функции mainprog(), а затем — роль объекта Sally.

В этом и есть основная польза области видимости. Вам не нужно помнить об именах объектов всех своих функций. Каждое имя действует только внутри своей области видимости.

Ниже показан еще один вариант программы (тоже не слишком удачный). В этом случае одно и то же имя Sal имеют глобальный и локальный объекты. Внутри функции JmpJack() области видимости обоих объектов перекрываются, но программа тоже будет работать правильно, поскольку локальный объект имеет приоритет над глобальным. Однако, создавая программу, в которой разные объекты имеют перекрывающиеся области видимости, вы можете легко запутаться, поэтому использовать одно и то же имя для обозначения разных объектов не следует.

В следующей инструкции задается область видимости объекта Sal (глобальная):

```
athlete Sal;
```

А здесь области видимости глобального и локального объектов Sal перекрываются:

```
void JmpJack(athlete Sal)
{
   Sal.up();
   Sal.ready();
}
```

В последнем примере показана область видимости объектов Sal и Sally:

```
void mainprog()
{
   athlete Sally;
   Sal.ready();
   JmpJack(Sal);
   Sally.readyO;
   JmpJack(Sally);
}
```



ПРИМЕЧАНИЕ Когда области видимости двух объектов или переменных, имеющих одинаковое имя, перекрываются, то используется более локальной объект. В одной области видимости нельзя объявлять объекты с одинаковыми именами.

Обычно новички забывают, что аргумент функции уже был объявлен в списке ее параметров. Например, в приведенном ниже примере гимнаст somebody объявлен в списке параметров. Поэтому объявлять этот объект снова нельзя:

```
void JmpJack(athlete sombody)
{
   athlete sombody;   // Неправильно! Объект somebody уже существует!
   ...
```

Примеры области видимости

В следующем примере показано неправильное объявление двух объектов с одинаковым именем. Первое объявление объекта Sal находится внутри функции mainprog(). Область видимости функции ограничивается закрывающей скобкой в конце списка ее инструкций. В этой области объявлять еще один объект Sal нельзя.

```
void mainprogO
{
   athlete Sal;
```

```
   jmpjack(Sal);
   athlete Sal;        // Ошибка! Первое объявление объекта Sal
   ...                 // находится в той же области видимости
;
```

Как будет видно в дальнейшем, другие конструкции C++ также ограничиваются фигурными скобками. В этом случае каждая пара открывающих и закрывающих скобок задает некоторую новую область видимости. Ниже приведен правильный пример. Второе объявление объекта Sal находится внутри новой области видимости. Этот новый объект будет существовать только во внутренней области видимости. Во всех обращениях к объекту Sal внутри нее вместо определенного первоначально объекта используется новый объект. Во всех же обращениях к объекту Sal до или после внутренних скобок используется исходный объект.

```
void mainprog()
{
   athlete Sal;
   Sal.ready()          // Использование первого объекта Sal
   {                    // Задание новой области видимости
      athlete Sal;
      athlete Sally;
      Sal.up();         // Использование нового объекта Sal
      Sally.left( );
   }                    // Конец внутренней области видимости
   Sal.right();         // Новое использование первого объекта Sal
}
```

В приведенном выше примере во внутренней области видимости был объявлен еще один объект Sally. Поэтому все обращения к объекту Sally возможны только внутри этой области видимости (напоминаем, что область видимости ограничена парой фигурных скобок). Все обращения к объекту Sally до внутренней открывающей скобки или после внутренней закрывающей скобки ошибочны.

Что нового мы узнали?

В этом уроке мы научились

- Писать функции.
- Использовать аргументы функций.
- Передавать аргументы по значению и по ссылке.
- Создавать заголовочные файлы.
- Задавать области видимости.



УРОК 5 Числа

- Хранение в переменных целых, чисел с плавающей точкой и алфавитно-цифровых символов
- Работа с числовыми переменными
- Ввод и вывод значений

В некоторых ситуациях в программах приходится использовать числа. В частности, с помощью чисел учитывают количество повторений, засекают время, а также представляют некоторые глобальные значения.

Как отмечалось в части I, *переменными* называют простые объекты. Переменные предназначены для хранения чисел.



ПРИМЕЧАНИЕ В названии *переменная* нашел отражение тот факт, что хранящееся в переменной значение меняется в процессе работы программы.

Числа и числовые переменные

Программа различает переменные по их именам. Правила работы с именами переменных те же, что и с именами объектов. *Переменная* — это ячейка (или набор ячеек) в памяти компьютера, где можно хранить число. Это число, в свою очередь, может обозначать буквы, цвета и вообще все, что мы захотим. В C++ имеются три основных типа переменных.

- *Целые.* В целых числах отсутствует дробная часть. Есть два типа целых — это типы int и long.
  - В переменной типа int могут храниться целые числа, значения которых лежат в интервале от −32768 до +32767.
  - В переменной типа Long могут храниться целые числа, значения которых лежат в интервале от -2147483648 до +2147483647.
- *Числа с плавающей точкой.* Это числа с дробной частью. Имеются два типа чисел с плавающей точкой — это типы float и double.
  - В переменной типа float могут храниться положительные и отрицательные числа с плавающей точкой, абсолютные значения которых лежат в интервале от 3.4×10^{-38} до 3.4×10^{38}. Количество значащих цифр ограничено.
  - В переменной типа double могут храниться положительные иотрицательные числа с плавающей точкой, абсолютные значения которых лежат в интервале от 1.7×10^{-308} до 1.7×10^{308}. Количество значащих цифр также ограничено.

О Символы. В переменной символьного типа char хранится целое число, представляющее собой код того или иного алфавитно-цифрового символа. Переменные типа char можно использовать в качестве целых переменных.

ПРИМЕЧАНИЕ Действительные интервалы значений для хранения переменной типа int или Long зависят от марки вашего компьютера.

Чтобы переменную можно было использовать в программе, она должна быть объявлена. Объявление переменной аналогично объявлению объекта. Как вы помните, объект объявляется указанием класса объекта, за которым следует его имя. Например:

```
athlete Sal;
Clock mytime;
```

Вы объявляете переменную, задавая ее тип, за которой указываете ее имя. Например:

```
int count, howmanytimes;
float elapsedtime;
```

Кроме вышеперечисленных типов целых чисел, имеется еще тип unsigned int — это так называемые беззнаковые целые. Беззнаковые целые могут быть только положительными. Зато при этом диапазон доступных значений удваивается. Например, следующая инструкция объявляет объект lapcount, в котором могут храниться целые значения в диапазоне от 0 до 65 535:

```
unsined int lapcount;
```

ПРИМЕЧАНИЕ Возможно, вы удивлены, что эти диапазоны представляют собой не круглые числа типа тысячи или миллиона. Это связано со спецификой хранения чисел в памяти компьютера.

Зачем нужны разные типы данных? Основная причина существования разных типов состоит в необходимости рационально использовать память компьютера. Переменная каждого типа предназначена для хранения числа определенного размера. Для хранения числа с плавающей точкой нам требуется места в четыре раза больше, чем для хранения буквы. Более того, мы не можем представлять числа с произвольным числом разрядов (значащих цифр), и, как будет ясно из дальнейшего, каждый тип имеет специфические ограничения.

ПРИМЕЧАНИЕ Все числовые типы удовлетворяют одним и тем же правилам синтаксиса.

Общие правила для числовых переменных

Числовые переменные подчиняются тем же правилам, что и все объекты;

- Для обозначения переменной используется идентификатор (имя).
- Правила использования имен переменных те же, что у объектов.
- Перед использованием переменная должна быть объявлена.
- Как и для любого объекта, при объявлении переменной перед именем переменной указывается ее тип.

ВНИМАНИЕ Объявление переменной вовсе не означает, что ей сразу присваивается какое-то определенное значение. Это означает лишь то, что вы можете использовать этот объект в своей программе. Например, когда объявляется переменная типа int, то компьютер всего-навсего выделяет место в памяти для хранения значения этого типа, но ни нуля, ни какого бы то ни было другого конкретного значения переменной не присваивается.

С переменной можно производить следующие действия:

- Проверить ее значение.
- Присвоить ей новое значение.
- Сравнить ее с другой переменной.

Значения

Рассмотрим следующий код:

```
#include "franca.h"
void mainprog()
{                    // Программа "раз, два, три"
   Athlete Sal;
   int one, two, three;
   Sal ready(5);
   Sal .say("one");
   Sal.say(one);
   Sal.say("two");
   Sal.say(two);
   Sal.say("three");
   Sal.say(three);
}
```

Что представляют собой значения one, two и three? Если вы запустите эту программу, то, возможно, заметите какие-то странные числа там, где Сэл будет пытаться сообщить вам значения целых переменных. Это происходит потому, что мы не придали нашим переменным никаких конкретных значений.



ВНИМАНИЕ В памяти компьютера всегда хранится какая-то информация, оставшаяся от предыдущих сеансов работы (если, конечно, компьютер не выключался). Поэтому, когда вы не присваиваете переменной никакого значения, в ней оказывается то, что осталось в соответствующей ячейке (или ячейках) памяти компьютера.

Присваивание значений

Для присваивания значения переменной используется оператор присваивания =. Оператор присваивания выглядит точно так же, как и хорошо знакомый вам из математики знак равенства. Чтобы присвоить какое-нибудь числовое значение, поставьте сначала имя переменной, затем оператор присваивания и наконец присваемое значение. Например, чтобы присвоить переменной howmany значение 21, необходимо написать:

```
howmany=21;
```

Конечно, переменную howmany сначала нужно объявить:

```
int howmany;
```

Очень важно понять, что в C++ символ = означает *присвоить следующее за ним значение,* то есть представляет собой действие, которое берет значение из правой части выражения и помещает его в переменную, имя которой находится в его левой части. Это в корне отличается от математического уравнения! Например, следующая инструкция в C++ неправильна, поскольку компилятор попытается разыскать значение howmany и поместить его в 21:

```
21=howmany;   // Такая инструкция никогда не будет работать!
```

Последнее лишено смысла, поскольку 21 не переменная и всегда остается числом 21!

Еще пример. Допустим, у вас в программе есть две целые переменные new и old. Предположим далее, что переменная new имеет значение 15, а переменная old — 13, Если выполнить следующую инструкцию, то оба объекта станут равными 13, поскольку компьютер возьмет значение справа от оператора присваивания и поместит его в переменную, имя которой находится слева от оператора:

```
new=old;
```

Напишем еще одну инструкцию:

```
old=new;
```

Если выполнить эти две инструкции, то оба объекта по-прежнему будут равняться 13, поскольку согласно первой инструкции, переменная new уже поменяла свое значение на 13.

Если вы хотите поменять местами значения переменных new и old, то с помощью следующей последовательности инструкций вам это не удастся, поскольку, когда вы копируете в переменную new содержимое переменной old, предыдущее содержимое переменной new стирается:

```
new=old;
old=new;
```

Чтобы правильно решить задачу, необходимо ввести дополнительную переменную, в которой сохранить первоначальное значение. Например:

```
temp=new;
new=old;
old=temp;
```

Инициализация

Переменной, как и любому объекту, можно присвоить значение сразу при ее объявлении. Например, следующая инструкция не только создает целую переменную maybe, но также присваивает ей значение 21:

```
int maybe=21;
```

Аналогичным образом вы можете инициализировать переменные в программе «раз, два, три», изменив соответствующее объявление следующим образом:

```
int one=1, two=2, three=3;
```

Можете теперь проверить программу.

Арифметические операторы в простых выражениях

Переменные также можно использовать в выражениях, содержащих арифметические операторы сложения (+) и вычитания (-), понимаемые в привычном нам смысле. Кроме того, переменные можно умножать (оператор *) и делить (оператор /), однако с этими операторами мы познакомимся несколько позже.

Целые переменные и числа можно объединять в выражения и получать результат, который затем с помощью оператора = можно сохранить в любой переменной. Например:

```
int maybe, hisage, difference; // Объявление целых объектов
```

```
maybe=21;                    // Сохранение значения 21
                             // в переменной maybe
difference=5;                // Сохранение значения 5
                             // в переменной difference
hisage=maybe+difference;     // Сложение значения переменной maybe
                             // со значением переменной difference
                             // сохранение результата
                             // в переменной hisage
```

В результате выполнения этого кода в переменной hisage окажется значение 26.

Инкремент и декремент

Вы можете также инкрементировать значения переменных (прибавлять к ним единицу). Для этого используется инструкция:

```
maybe=maybe+1;
```

Здесь компьютер берет текущее значение переменной maybe, прибавляет к нему единицу и сохраняет результат в той же переменной. (Видите, насколько это отличается от математического уравнения?) Аналогично можно декрементировать значение переменной с помощью следующей инструкции:

```
maybe=maybe-1;
```

Операторы ++ и —

Для выполнения операций инкремента и декремента в C++ имеется еще одна возможность. Она реализуется с помощью соответствующих операторов инкремента (++) и декремента (-•). Например:

```
maybe++;
```

У выражения с оператором инкремента есть одно важное свойство: оно может использоваться внутри другого выражения:

```
maybe=25;
hisage=maybe++;
```

В этом случае сначала используется исходное значение maybe, а затем это значение инкрементируется. Поэтому после выполнения этих инструкций значения переменных будут следующими:

- Значение переменной maybe будет равняться 26, поскольку оно было инкрементировано.
- Значение переменной hisage будет равняться 25, поскольку в ней было сохранено исходное значение переменной maybe.

Если вместо предыдущей последовательности инструкций мы напишем другую, то сначала будет инкрементирована переменная maybe, а затем ее значение присвоено переменной hisage:

```
maybe=25;
hisage=++maybe;
```

Таким образом, значения обеих переменных (maybe и hisage) окажутся равными 26.

В качестве третьего варианта рассмотрим следующий фрагмент кода:

```
maybe=25;
hisage=++maybe++;
```

Как вы думаете, чему в результате будут равны значения переменных hisage и maybe? Ведь переменная maybe инкрементируется дважды, до и после выполнения операции присваивания.



ПРИМЕЧАНИЕ Оператор декремента -- функционирует аналогично.



СОВЕТ Операторы ++ и -- можно размещать как до, так и после переменной. В первом случае переменная сначала инкрементируется (декрементируется), а уже затем используется в выражении,

Самостоятельная практика

Допустим, что в программе имеется следующее объявление:

```
int i=1, j=2, k=3;
```

Используя приведенные ниже выражения, определите значения переменной k после выполнения каждого из них. Если выражение записано неверно, объясните почему. Считайте, что все инструкции выполняются последовательно.

```
k=++k+j;
k=i+j++;
k=i-j;
i+j=k;
k++=j;
```

Операторы отношения

С помощью операторов отношения можно сравнивать значения. В общем случае, производя сравнение, вы пытаетесь узнать, является ли данное значение:

- == равным другому.
- != не равным другому.
- < меньшим другого.
- <= меньшим или равным другому.
- > большим другого.
- >= большим или равным другому.

Сравнения полезны для управления количеством повторений, а также для принятия программных решений (о том и другом мы узнаем позднее).



ПРИМЕЧАНИЕ В большинстве случаев обращаться с объектами и переменными мы будем одинаково. Тем не менее небольшая разница между ними все-таки есть. На заре компьютерной эры существовали только переменные. В них хранилась информация и с ними можно было производить некоторые арифметические операции. Объекты придумали недавно. Кроме хранения информации в объектах, можно определить виды операций, которые будут выполняться с объектами. Другими словами, можно задать те виды сообщений, на которые будет реагировать ваш объект. Таким образом, объект это нечто вроде «умной» переменной, поскольку его можно обучить различным операциям.

Ввод значений

Очень часто необходимо сообщать компьютеру ту или иную информацию. Это может быть *ввод* (input) с клавиатуры, с дискеты или с какого-либо датчика (например, часов). Благодаря вводу информации компьютер может реагировать на различные ситуации.

Вообще говоря, ввод происходит только тогда, когда этого требует программа. Введенная информация всегда присваивается переменной или объекту, так что программа может ее проверить и использовать.

На этом уроке мы изучим только ввод информации с клавиатуры. Ввод с дискеты обсуждается на уроке 23. Описываемые методы ввода информации доступны только при использовании программного обеспечения книги, которое можно найти на прилагаемой дискете.



ПРИМЕЧАНИЕ Об основных приемах ввода и вывода информации в C++ рассказывается в части VIII.

Можно сделать так, чтобы программа выдавала запрос на ввод значения с клавиатуры, а затем присваивала его некоторой переменной. Так, например, если мы хо-

тим научить компьютер рассчитывать сдачу, было бы глупо создавать для каждой покупки новую программу. Вместо этого лучше вводить в компьютер цену покупки и уплаченную за нее сумму.

Рассмотрим два метода ввода значений:

- Метод ask доступен только при наличии программного обеспечения книги.
- Метод Cin также доступен только при наличии программного обеспечения книги, но он очень похож на стандартный метод ввода информации в C++.

Функция ввода ask()

Функция ввода ask() создает диалоговое окно, которое предлагает пользователю ввести значение с клавиатуры. Это значение потом присваивается какой-нибудь переменной, и с ним в дальнейшем можно работать. Синтаксис функции ask() имеет следующий вид:

```
ask(запрос);
```

Здесь параметр запрос — это предложение, предлагающее пользователю ввести информацию. Например, следующая инструкция вызовет появление диалогового окна (рис. 5.1), в котором пользователю предлагается ввести в текстовое поле цену покупки:

```
Price=ask("Please enter the price:");
```

После ввода нужного значения и щелчке на кнопке OK (или нажатии клавиши Enter), диалоговое окно исчезнет и введенное значение будет присвоено переменной Price.

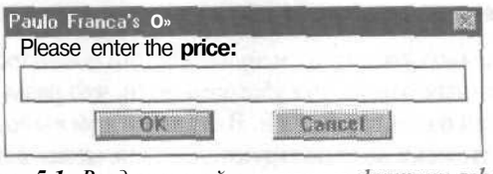


Рис. 5.1. *Ввод значений с помощью функции ask()*

ПРИМЕЧАНИЕ Кнопка Cancel в этом диалоговом окне не функционирует.

Имитация входного потока объектом Cin

Альтернативный способ ввода значений предлагает объект Cin, имитирующий стандартный ввод информации в C++. Синтаксис в этом случае отличается от синтаксиса функции ask();

```
Cin >> имя_переменной;
```

Параметр имя_переменной — это имя переменной, в которой вы собираетесь сохранить введенное значение. Главное неудобство этого метода в том, что пользователю не выдается никакого приглашения на ввод информации. Однако перед вводом значения можно сначала вывести текст такого приглашения (см. следующий раздел о выводе значений).

Эффект от использования объекта Cin аналогичен тому, который дает функция ask(). Также выводится диалоговое окно, предлагающее пользователю ввести целое число с плавающей точкой или последовательность символов. (рис. 5.2).

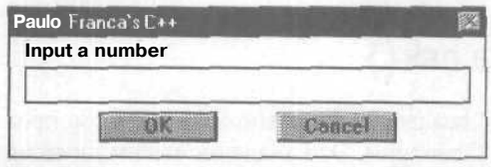


Рис. 5.2. *Ввод значений с помощью объекта Cin*

Вывод значений

Аналогично тому, как в определенных ситуациях возникает необходимость вводить в компьютер информацию из внешнего мира, иногда вы сами просите компьютер сообщить вам те или иные данные. Информация, которую предлагает компьютер, называется *выводом* (output).

Вывод может быть реализован в виде чисел или символов на экране или принтере, записи на дискете, картинки на экране, звуков, издаваемых специальным устройством, и т. д. С некоторыми из этих методов вывода информации мы уже познакомились. В данном разделе мы ограничимся выводом на экран чисел или символов.

Вы уже знаете, как выводить на экран значения или сообщения с помощью гимнастов и информационных рамок. Вы уже заметили, что размер выводимого в этих случаях предложения очень ограничен. В стандартном выводе C++ окна (в нашем случае рамки) практически не используются, тем не менее наши рамки очень похожи на окна, с которыми вы столкнетесь, перейдя от стандартного текстового интерфейса к графическому.

Для программируемого вывода информации на экран используется объект Cout, аналогичный уже знакомому нам объекту Cin. Объект Cout также поставляется вместе с программным обеспечением книги и служит для имитации стандартного вывода информации в C++.

Имитация выходного потока объектом Cout

С помощью объекта Cout вывод информации выполняется следующим образом. В нижней части экрана формируется рамка, в которой появляются выводимые

данные. Поскольку рамка всего одна, вывод новой порции информации стирает предыдущую (рис. 5.3).

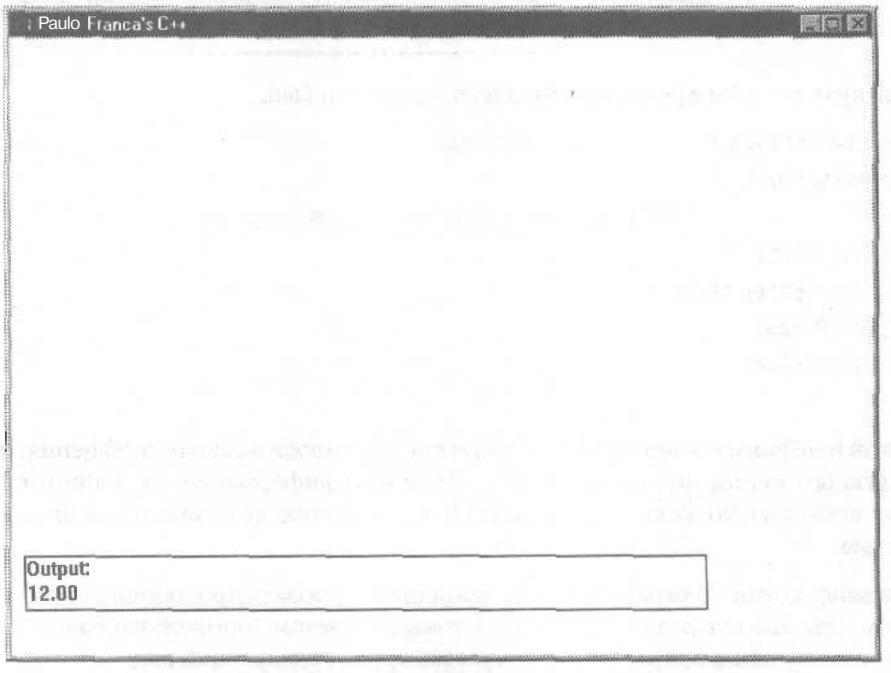


Рис. 5.3. *Вывод информации с помощью объекта Cout*

Синтаксис объекта Cout аналогичен синтаксису объекта Cin, только направление стрелок меняется на противоположное. Это поможет вам запомнить, что значения идут *из* переменной *в* объект Cout и *из* объекта Cin в переменную.

```
Cout << имя_переменной;
```

С помощью следующей инструкции можно вывести значение переменной Price:

```
Cout << Price;
```

В представленной ниже программе для ввода информации используется функция ask(), а для вывода — функция say():

```
#include "franca.h"          // c2input.cpp
athlete Julia;
void mainprog()
{
   float Price;
   Julia.ready();
   Price=ask("Please enter the price:");
   Julia.say(Price);
}
```

ПРИМЕЧАНИЕ Стандартный ввод и вывод в C++ выполняется соответственно с помощью объектов cin и cout (с маленькой буквы), о которых рассказывается в части VIII.

В следующем примере используются объекты Cin и Cout

```
ftinclude"franca.h"          // c2inout.cpp
void mainprog()
{
   float Price;
   Cout<<"Enter the price:";
   Cin>>Price;
   Cout<<Price;
}
```

В этой программе объект Cout используется для вывода на экран сообщения, предлагающего пользователю ввести определенную информацию (в данном случае цену покупки), поскольку сам объект Cin к сожалению не обладает такой возможностью.

С помощью этой программы можно проиллюстрировать простейшие выражения. Допустим, мы хотим вычислить цену товара с учетом торговой надбавки. После соответствующих исправлений программа примет следующий вид:

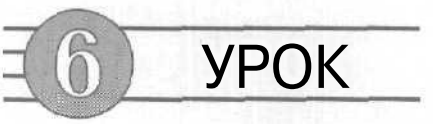
```
#include "franca.h"
void mainprog()
{
   float Price, tax=8.25;          // Новая переменная
   Cout<<"Enter the price:";
   Cin>>Price;
   Price=Price+(Price*tax/100);   // Расчет новой цены
   Cout<<Price;
}
```

В этой программе новая переменная tax, инициализированная значением 8.25, что представляет выраженную в процентах торговую наценку. После считывания значения переменной Price программа вычислит новое значение цены с учетом наценки и снова поместит это значение в переменную Price.

Что нового мы узнали?

В этом уроке мы научились

- Оперировать с числовыми переменными.
- Вводить значения с клавиатуры.
- Выводить значения на экран.



Решение проблем с помощью функций

- Возвращаемые значения функций
- Встраиваемые функции
- Решение проблем

Теперь, когда вы уже знаете, как обращаться с числовыми переменными, функции могут принести вам гораздо больше пользы. На этом уроке мы продолжим рассмотрение способов создания и использования функций, а потом снова вернемся к решению проблем. Сейчас вы уже обладаете достаточными знаниями, чтобы с помощью функций существенно упростить программные решения ваших задач.

В реальной жизни вряд ли кто-нибудь может от вас потребовать специально создавать функцию для решения той или иной проблемы. Перед вами будет поставлена задача, и вы должны будете решить ее наилучшим образом, используя при этом то, что сочтете нужным. Фактически, большинство проблем, с которыми мы сталкиваемся в реальной жизни, вообще должным образом не определены!

Ваш клиент может сам до конца не понимать, что он хочет (и даже вообще не догадываться о реальной проблеме). Именно вам придется отвечать за изучение проблемы и нахождение решения, которое устроит вас и вашего клиента.

В решении задач функции играют важнейшую роль. Их использование в программах дает нам следующие преимущества:

- Функции упрощают тело основной программы.
- Благодаря функциям легче разработать, понять и модифицировать программу.
- Для решения тех или иных задач, составляющих проблему в целом, подходящие функции можно запасти заранее.

Старайтесь пользоваться тем, что имеете под рукой, поскольку на создание и отладку новых фрагментов программы требуется определенное время. Если у вас нет готовых функций, создавайте новые с таким расчетом, чтобы позднее их можно было использовать для решения схожих проблем. Так вы сэкономите кучу времени. Многие проблемы, возникающие в реальной жизни, часто похожи друг на друга.

Вы сами должны решить, насколько активно следует пользоваться функциями. Обычно необходимо сделать следующее:

- Разработать программу для решения поставленной задачи.
- Сделать ее как можно короче.
- Обеспечить возможность ее многократного использования.
- Вставить в программу все, что сумеете найти готового.
- Максимально упростить возможность понимания и модификации программы.
- Сдать программу к назначенному сроку.
- Максимально сократить количество программных ошибок.

Кроме того, очень важно, чтобы все ваши программы и функции были документированы. Снабдите программу разумным числом комментариев и при необходимости документацией (типа пользовательского руководства).

Возвращаемые значения функций

Обычно каждый вызов функции ведет к тому или иному результату. Результатом может быть анимация на экране (как было с функциями JmpJack()), значение (как было при определении времени в объекте Clock) или некоторая их комбинация.

Средства связи с функциями обеспечивают параметры. Параметры можно использовать для:

- Передачи информации в функцию (как было с функциями JmpJack()).
- Вывода информации из функции (в этом случае параметр должен передаваться по ссылке).

В качестве иллюстрации рассмотрим функцию вычисления разности двух чисел с плавающей точкой. Например, пусть по заданной стоимости товара и уплаченной сумме функция рассчитает сдачу. Такая функция должна оперировать с тремя значениями:

- Стоимостью товара, переменная theprice (известна).
- Уплаченной суммой, переменная theamount (известна).
- Требуемой сдачей, переменная thechange (рассчитывается функцией).

Можно написать функцию с тремя этими параметрами. Функция должна рассчитывать разность между первыми двумя параметрами, а результат размещать в третьем. Помните, этот параметр должен передаваться по ссылке. С помощью первых двух параметров реализуется ввод данных в функцию, а с помощью третьего — вывод.

Такая функция не запрашивает ввод с клавиатуры и ничего не выводит на экран. Тем не менее с помощью параметров theprice и theamount в функцию вводятся стоимость покупки и уплаченная сумма, а с помощью параметра thechange полученный результат возвращается в основную программу.



ПРИМЕЧАНИЕ Ввод с клавиатуры и вывод на экран выполняется в функции mainprog().

На C++ эту процедуру можно выполнить с помощью следующего фрагмента кода:

```
void dif(float theprice, float theamount, float &thechange)
{
   thechange=theamount-theprice;
}
```

В этом случае фрагмент программы с переменными amount и price, в котором рассчитанная сдача выводится на экран, будет выглядеть так:

```
...
dif(price, amount, &change);
Cout<<change;
...
```

Имеется и другой способ отображения результата выполнения функции на экране:

```
Cout<<dif(price, amount, change);
```

Такой синтаксис подходит только для функций, возвращающих в качестве результата не ссылку, а значение. В некоторых языках программирования функции, которые возвращают значение, отличают от тех, которые его не возвращают. Например, в языке Паскаль функции, которые не возвращают значений, называются процедурами.

Как компьютер узнает, возвращает функция значение или нет? Почему, например, нельзя написать инструкцию:

```
Number=JumpJack(Sal);
```

Но в то же время *допустима* следующая инструкция:

```
Price=ask("Please enter the price");
```

Чем эти функции отличаются друг от друга?

Типывозвращаемыхзначений

В C++ искомое отличие находится в заголовке функции — первой строке, в которой функции назначается имя. Например, рассмотрим следующий заголовок функции:

```
void diff(float theprice, float theamount, float &thechange)
```

При определении функции первым определяется тип ее возвращаемого значения. Обратили ли вы внимание на ключевое слово void перед именами всех рассматривавшихся до сих пор функций? Оно означает, что функция не возвращает значение.

Чтобы функция возвращала значение, необходимо:

- В заголовке функции определить тип возвращаемого значения. Если он отличен от void, то функция должна возвращать результат. Если тип возвращаемого значения не указан, в C++ это равносильно заданию типа int.
- Используя ключевое слово return, включить в тело функции хотя бы одну инструкцию, определяющую возвращаемый результат.

Например, рассмотрим случай, в котором модифицированная функция dif() должна возвращать разность двух чисел с плавающей точкой, переданных ей в качестве

аргументов. Поскольку возвращаемым значением тоже должно быть число с плавающей точкой, заголовок функции записывается следующим образом:

```
float dif(float value1, float value2)
```

Вот и весь синтаксис. Заголовок показывает компилятору, что функция с именем dif имеет два аргумента value1 и value2 (оба являются числами с плавающей точкой) и возвращает результат, который также является числом с плавающей точкой. Теперь третий параметр для хранения в нем результата больше не нужен.

Чтобы закончить формирование тела функции, необходимо знать синтаксис возвращения значений. Он очень простой и состоит из ключевого слова return, за которым следует какое-либо выражение:

```
return выражение;
```

Вычисленное значение данного выражения и является возвращаемым значением функции. Теперь нашу функцию можно полностью закончить:

```
float dif(float value1, float value2)
{
   return value1-value2;
}
```



ВНИМАНИЕ После выполнения инструкции с ключевым словом return функция завершается! Инструкции, расположенные в функции следом за инструкцией return, выполнены не будут.

Использование этой функции иллюстрируется приведенной ниже программой c2change.cpp. Программа запрашивает у пользователя информацию о стоимости товара и уплаченной сумме, а затем рассчитывает сдачу.

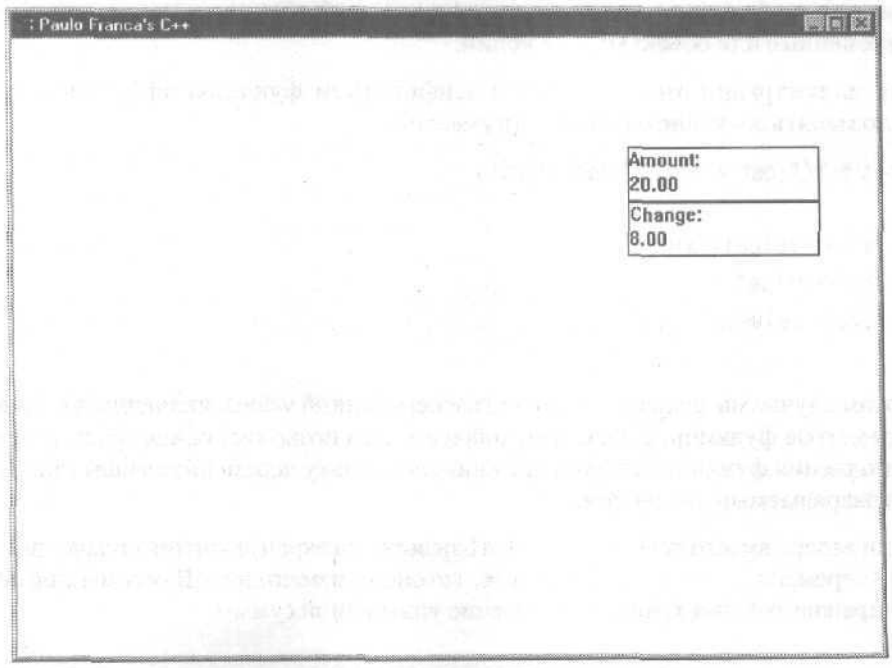
Конечно, в данном случае проще вычислить значение разности без всяких функций внутри основной программы. Однако даже этот простой пример пополнит наши знания о функциях и поможет нам почувствовать их достоинства.

Алгоритм программы очень прост:

- Объявить переменные price, amount и change.
- Запросить у пользователя значение цены.
- Запросить у пользователя значение уплаченной суммы.
- Вычислить сдачу (с помощью функции dif()).
- Сообщить пользователю значение сдачи.

Далее приведена программа c2change.cpp (результат выполнения которой показан на рис. 6.1):

```
tfinclude "franca.h"          //c2change.cpp
float diff(float value1, float value2)
{
   return value1-value2;
}
void mainprog()
{
   float price, amount, change;
   Box given("Amount:"), thechange("Change:");
   price=ask("What is the price to pay?");
   amount=ask("How much are you giving me?");
   change=dif(amount, price);
   given.say(amount);
   thechange.say(change);
}
```



Рис, 6.1. *Результат выполнения программы c2change.cpp*

Обратите внимание, что после запроса значений для переменных price и amount мы использовали выражение:

```
change=dif(amount, price);
```

Значение этого выражения вычисляется с помощью функции dif(). Как мы уже знаем, эта функция возвращает значение разности первого и второго аргументов. Затем это значение присваивается переменной change.

В приведенном выражении аргумент price расположен после аргумента amount. В функции dif() параметры value1 и value2 расположены в том же порядке. Когда во время выполнения программы вызывается функция dif(), параметр value1 заменяется значением переменной amount, а параметр value2 — значением переменной price. Затем эти значения используются в выражении, а результат становится возвращаемым значением функции.

Помните, что соответствие между аргументами определяется только порядком их расположения. Следующее выражение ведет к совершенно неправильному результату:

```
change=dif(price,amount);
```

Оригиналы и копии

При вызове функции с параметрами функция обрабатывает не сами аргументы (переменные или объекты), а их копии.

Для иллюстрации этой ситуации модифицируем функцию dif(), чтобы можно было менять значение одного из аргументов:

```
float dif(float value1, float value2)
{
   value1=value1-value2;
   Cout<<value1;
   return value1;
}
```

В этом случае мы сохраняем разность в переменной value1, являющейся одним из параметров функции. Увидеть это значение нам позволяет объект Cout. Результат выполнения функции остается прежним, поскольку переменная value1 становится ее возвращаемым значением.

Если теперь вместо переменной value1 вывести на экран значение уплаченной суммы (переменную amount), вы увидите, что оно не изменилось. В переменной amount по-прежнему будет храниться значение уплаченной суммы.



СОВЕТ Этот пример вывода информации приведен исключительно в качестве иллюстрации. Поскольку возвращаемое значение функции dif() передается в основную функцию mainprog(), именно в ней и нужно было выводить это значение на экран. Без особой необходимости не стоит выполнять вывод информации внутри функций, поскольку это ограничивает возможность их повторного использования.

Встраиваемые функции

Каждый раз, когда ваша программа вызывает функцию, компьютеру требуется определенное время, чтобы сначала переключиться на функцию, а затем вернуться обратно в программу. Главным образом это время тратится на то, чтобы скопировать в параметры функции значения, хранящиеся в ее аргументах.

В любом языке программирования для выполнения не слишком сложных задач создавать функции не всегда разумно. Это связано с тем, что время, которое тратит компьютер переключаясь на выполнение функции и возвращаясь обратно, может в значительной степени свести на нет преимущества функций.

Отличительной чертой языка C++ является то, что он допускает особый тип функций, код которых размещается в месте их вызова. Это так называемые *встраиваемые функции* (inline functions). Другими словами, там, где в вашей программе записан вызов такой функции, компилятор просто делает копию всех ее инструкций. Таким образом, программе вообще не приходится вызывать функцию, поскольку в соответствующее место программы компилятор вписывает ее полный код.

Однако не все функции могут быть встраиваемые. Имеется несколько ограничений, главное из которых состоит в том, что внутри встраиваемых функций нельзя использовать циклы. (О циклах вы узнаете на уроке 7.)

Для обозначения встраиваемой функции в ее заголовок включается спецификатор `inline`. Тем не менее не забывайте, что эта особенность относится только к языку C++ и недоступна в других языках программирования.

С примером встраиваемой функции мы познакомимся в следующем разделе, где вновь вернемся к проблемам, рассмотренным на «Первом этапе великого похода» из урока 2. С помощью функций мы попробуем упростить решение этих проблем.

Второй этап великого похода

Функции сильно упрощают роботам закрашивание квадратов. Действительно, насколько легче становится задача, если робот уже знает, как рисовать линию!

При наличии функции рисования линии задача рисования квадрата может быть представлена в виде следующего алгоритма:

Рисование линии

Поворот направо

Рисование линии

Поворот направо

Рисование линии

Поворот направо

Рисование линии

Поворот направо

Последний поворот хотя и не обязателен, но его нужно выполнить, чтобы в конце концов робот оказался в исходном положении. Для рисования линий и квадратов воспользуемся нашим знакомым роботом (объектом Tracer).

Поскольку циклов мы еще не изучали, будем рисовать линии последовательно. Алгоритм достаточно очевиден.

Рисование линии размером в 3 ячейки:

Закрашивание ячейки

Шаг вперед

Закрашивание ячейки

Шаг вперед

Поскольку робот закрашивает только те ячейки, которые находятся прямо перед ним, полная длина линии, включая начальную и конечную ячейки, равна трем. Но при этом первая ячейка остается чистой. Если это доставляет вам беспокойство, всегда можно исправить ситуацию.

Рисование линии размером в 3 ячейки, включая первую ячейку:

Закрашивание ячейки

Шаг вперед

Поворот направо

Поворот направо // Поворот обратно

Закрашивание ячейки // Закрашивание первой ячейки

Поворот направо

Поворот направо // Поворот обратно

Закрашивание ячейки

Шаг вперед

Теперь легко написать функцию реализации любого из описанных выше алгоритмов:

```
Robot Tracer;
void line3()
{
   Tracer mark();
   Tracer step();
   Tracer mark();
   Tracer stepC);
}
```

Независимо от того, для чего вы создаете свою функцию, старайтесь делать ее возможно более полезной. Тогда в дальнейшем вы снова сможете ею воспользоваться.

Например, функция была бы гораздо полезнее с возможностью задания длины линии. Этим мы займемся позже, когда освоим циклы. Но тем не менее вы и сейчас можете несколько улучшить эту функцию, если добавите в нее параметр выбора цвета:

```
void line3(int color)
{
  Tracer mark(color);
  Tracer step();
  Tracer mark(color);
  Tracer step();
}
```

Возможно, вас раздражает то, что вам придется указывать цвет при каждом обращении к функции? Нет проблем! В заголовке функции можно задать значение цвета по умолчанию:

```
void line3(int color=2)
```

С помощью функции рисования линий можно легко рисовать квадраты или создать для этого специальную функцию. При создании функций старайтесь сохранять решения наиболее общих задач. Это *обязательно* когда-нибудь вам пригодится.

Ниже представлен один из возможных вариантов функции рисования квадрата:

```
void square3(int color=2)
{
  line3(color);
  Tracer.rlghtC);
  line3(color);
  Tracer.right();
  line3(color);
  Tracer.right();
  line3(color);
  Tracer.right();
}
```

Использование функций

Даже при решении несложной задачи рисования квадрата есть несколько вариантов применения функций:

- С использованием двух функций line3() и square3(). При этом функция square3() изображает квадрат с помощью функции line3(), которая рисует каждую линию квадрата.
- С использованием трех функций point(), line3() и square3(). Функция point() могла бы закрашивать ячейку и делать шаг вперед, это упростило бы код функции line3().
- Без использования функции line3(). При этом функция square3() сама рисовала бы каждую линию.
- Вообще без использования функций. При этом все инструкции рисования квадрата оказались бы в основной функции mainprog().

Выбор оптимального варианта

На вопрос, какой вариант лучше, нет однозначного ответа. При его решении нужно учитывать несколько факторов. Кроме того, ответ во многом зависит от личности отвечающего.

Очевидно, что последний вариант — не использовать функции вообще — самый неудачный. Программисты, которые не пользуются функциями, обычно тратят на создание и отладку программ гораздо больше времени, чем другие. На это есть несколько причин:

- Код программы становится более сложным.
- Теряется возможность многократного использования ранее разработанных фрагментов программного обеспечения.

Некоторые программисты утверждают, что чем меньше в программе функций, тем быстрее она работает. Вообще говоря, это действительно так. Каждый вызов функции требует дополнительного времени. Тем не менее, как правило, для большинства приложений это не столь критично. Во многих ситуациях гораздо удобнее быстро закончить программу и иметь возможность, в случае необходимости, легко ее изменить, чем корпеть над высокоскоростным вариантом, который обычно приходится долго отлаживать и в дальнейшем почти невозможно модернизировать.

Второй вариант — использование функций на трех уровнях — кажется явным перебором. Функции, решающие слишком мелкие задачи, могут стать самоцелью и отнимать массу компьютерного времени. Иногда это существенно, иногда нет. С другой стороны, как программисты вы должны понимать, что код такой программы будет содержать минимально возможное число строк. Вот уж идеальный вариант для лентяя.

А плохо ли быть ленивым программистом? Ни в коем случае! Согласитесь, что наиболее компактный код занимает особое место среди всех возможных вариантов решения задачи. Что же в этом может быть плохого?

Как я уже говорил, вопрос о том, следует ли миллион раз вызывать одну и ту же функцию для выполнения несложного задания, является весьма спорным. Слиш-

ком много времени уходит не на реальную работу, а на всевозможные переключения. В результате напрашивается естественный вывод: такого использования функций лучше избегать.

Это абсолютно справедливо для большинства языков программирования, но не для C++. Помните встраиваемые функции?

Небольшие фрагменты кода, в которых отсутствуют циклы, — чем не прекрасные кандидаты на роль встраиваемых функций.

Программа c2squar2.cpp иллюстрирует вариант использования всех трех функций. В функции mainprog() робот делает пять шагов на восток, затем семь шагов на юг и начинает рисовать. В результате получаются два разноцветных квадрата.

```
#include "franca.h"          //c2square2.cpp
Robot Tracer;
inline void paint(int color=2)
{
   Tracer.mark(color);
   Tracer.step();
}
void line3(int color=2);
{
   paint(color)
   paint(color)
}
void square3(int color=2);
{
   line3(color);
   Tracer.right();
   line3(color);
   Tracer.right();
   line3(color);
   Tracer.right();
   line3(color);
   Tracer.right();
}

void mainprog()
{
   Tracer.face(3);
   Tracer.step(5);
   Tracer.right();
   Tracer.step(7);
   Tracer.left();
   square3();
   Tracer.step(5);
   square3(3);
}
```

Правила и соглашения

На основе пройденного материала попытаемся систематизировать правила синтаксиса.

Определение функции

Определением функции (functiondefinition) называется та часть ее программного кода, в которой описаны действия, выполняемые при вызове функции. В определение входит заголовок и тело функции. В общем случае определение функции выглядит следующим образом:

```
заголовок функции
{
   тело функции
}
```

Заголовок функции

Заголовок функции (function header) содержит:

- Тип возвращаемого значения (для функций без возвращаемого значения указывается тип void).
- Имя (или идентификатор) функции, а также список параметров и их типов, заключенный в круглые скобки (если параметров нет, то пространство внутри скобок остается пустым).



ПРИМЕЧАНИЕ Вы также можете встретить ключевое слово void внутри круглых скобок. Например, инструкция float time (void) означает то же, что и пустые скобки, то есть float time().

Если параметров несколько, то они разделяются запятыми и перед каждым из них должен быть указан его тип (или класс). Этот список служит для объявления параметров.

Например, следующий заголовок функции

```
voidJmpJack(athletesomebody)
```

означает, что:

- У функции нет возвращаемого значения (void).
- Имя функции JmpJack.
- У функции имеется один параметр somebody типа athlete (следовательно, внутри функции нельзя объявить другой объект somebody типа athlete).

Тело функции

Тело функции (function body) включает в себя код, выполняемый при вызове функции. Это просто фрагмент программного кода, границы которого обозначены фигурными скобками, содержащий одну или несколько инструкций.

Например, если тело функции JmpJack() имеет следующий вид:

```
somebody.up();
somebody.ready();
```

то функция в целом будет такой:

```
void JmpJack(athlete somebody)
{
   somebody.up();
   somebody.readyO;
}
```

Здесь уместно напомнить следующие правила:

- Если некоторые аргументы имеют значения по умолчанию, они должны находиться в конце списка аргументов.
- Нельзя вызвать функцию раньше, чем она определена!

Соответствие типов

При вызове функции должно быть обеспечено *соответствие типов* (type matching) ее аргументов и параметров. Например, если функция определена следующим образом, то при вызове функции первый аргумент должен быть типа Clock, а второй — типа athlete:

```
void jumps(Clock timekeeper, athlete somebody)
```

Компилятор проверяет соответствие между типами аргументов и параметров, а при отсутствия такового выдает сообщение об ошибке.

Перегрузка функций

В отличие от других языков программирования, C++ допускает использование нескольких функций с одинаковыми именами при условии различия в числе или типе их аргументов. Это называют *перегрузкой функций* (function overloading).

Например, в одной программе могут быть две функции JmpJack(), первая с одним, а вторая с двумя аргументами. Однако больше никаких функций JmpJack() с одним или двумя параметрами класса athlete в программе быть не может.

Компилятор отличит друг от друга следующие два вызова и подставит в каждом случае требуемую функцию:

```
JmpJack(Sal);
JmpJack(Sal, Sally);
```

Значения и ссылки

По умолчанию аргументы передаются в функцию *по значению*. Это означает, что при вызове функции создается копия ее аргумента (объекта или переменной), которая затем используется функцией. Таким образом, при изменении параметра функции ее аргумент не меняется.

Если в заголовке функции перед именем переменной поставить символ амперсанда &, то во всех выполняемых функцией операциях будет использоваться исходная переменная, а не ее копия.

Когда передаваемый в функцию объект достаточно сложен, *передача по ссылке* может ускорить работу программы. Экономия времени достигается за счет того, что отпадает необходимость в копировании объекта. В том случае аргументом функции не может быть константа (то есть число). Например, функцию:

```
float dif(float &value1, float &value)
```

Нельзя вызывать следующим образом;

```
change=dif(100.00, price);
```

Благодаря передаче по ссылке функция может менять значение переменной value1, но изменить значение константы 100.00 невозможно. Такая попытка (как в предыдущей инструкции) приведет к сообщению об ошибке.

Вызов функций

Чтобы выполнить в программе некоторый набор действий, описанный в определении функции, можно использовать *вызов функции* (function call). В инструкцию вызова функции входит ее имя и в скобках список ее аргументов:

```
имя_функции (список_аргументов);
```

У функции с параметрами в скобках должны быть перечислены все ее аргументы. Если функция получает несколько аргументов, то соответствие между реальным объектом и параметром определяется только порядком их следования.

Прототипы функций

Иногда функцию приходится использовать до ее определения. Поскольку компилятор анализирует программный код последовательно, он может решить, что вы пытаетесь вызвать несуществующую функцию. Если вы сталкиваетесь с этой проблемой, вам надо известить компилятор, что функция с параметрами и возвращаемым значением заданного типа существует. Это означает, что вам необходимо задать *прототип функции* (function prototype).

ВНИМАНИЕ Точка с запятой после заголовка функции является весьма распространенной ошибкой, которая заставляет компилятор думать, что заголовок — это всего лишь прототип функции. В случае такой ошибки компилятор не может найти определения функции.

Прототип функции очень похож на ее заголовок, за исключением следующих отличий:

О В конце прототипа функции ставится точка с запятой:

```
float change(float price, float amount);
```

О Имена параметров можно опустить, оставив только их типы:

```
float change(float, float);
```

Прототипы функций не требуются, пока нет необходимости сообщать компилятору о функции, которая в программе определяется позднее. Тем не менее их включение в программу без необходимости ошибкой не считается.

Исключение лишних заголовочных файлов

Включая в программу заголовочные файлы с помощью директивы #include, вы можете случайно скопировать один и тот же файл несколько раз. Это бывает, когда в программе имеется несколько директивtfincLudeили когда в файле, который вы включаете в свою программу, тоже есть директива #include, которая, в свою очередь, уже включила в программу запрошенный вами файл.

Например, пусть в вашу программу включается программа gymnast.h, в которую, в свою очередь, включается программа JmpJack.h. В этом случае, как показано ниже, в программе gymnast.h тоже имеется директиватfincludeдля заголовочного файла JmpJack.h.

Ваша программа:

```
ftinclude "gymnast h"
ttinclude"JmpJack.h"
```

Программа gymnast.h:

```
#include"JmpJack.h"
```

Если вы хотите все сделать правильно, необходимо исключить лишнее копирование файла JmpJack.h. Для этого в начало и конец заголовочного файла добавляется несколько строк. В начало добавляются две строки:

```
ffifndef  JUMPJACK_H
#define JUMPJACK_H
```

А в конец одна строка:

```
ttendif
```

Параметр JUMPJACK_H в каждом файле заменяется соответствующим именем.

Инструкции, которые начинаются с символа фунта #, не относятся к языку C++. Это инструкции той части компилятора, которая называется препроцессором. Они также называются директивами препроцессора.

Попрактикуйтесь в синтаксисе директив препроцессора и напишите фрагмент кода с директивой #include, чтобы дважды включить в программу заголовочный файл JmpJack.h. Затем вставьте в программу рассмотренные выше директивы препроцессора для файла JumpJack.h и выполните программу снова.

После того как вы справитесь с этим заданием, напишите заголовочный файл, содержащий функции подскока и поворота влево-вправо. Параметрами этих функций должны быть объекты типа athlete. Назовите этот заголовочный файл fitness.h и сохраните его на своем компьютере.

Кроме того, можете попробовать написать программу с использованием заголовочного файла fitness.h и знакомых нам гимнастов Сэла и Салли. Пусть в этой программе они выполнят по четыре подскока и по три поворота.

Целые числа и числа с плавающей точкой

Целая переменная может хранить целые числа в интервале от -32 768 до +32 767. Такие переменные объявляются с помощью ключевого слова int перед идентификатором соответствующей переменной. Числа с дробной частью хранятся в переменных типа float, которые, в свою очередь, объявляются с помощью ключевого слова float. В одной инструкции можно объявить несколько переменных одного типа. Например:

```
int number, alpha, count;
float x, y, z;
```

При объявлении переменных любой из них можно присвоить начальное значение. Например;

```
int one=1, two=2, three=3, other;
float x=3.45, y, z;
```

В первой инструкции объявляются четыре целые переменные: one, two, three и other, причем первые три инициализируются соответственно значениями 1, 2 и 3, а четвертая не инициализируется. Аналогично, переменная с плавающей точкой x инициализируется значением 3.45, а переменные y и z объявляются, но не инициализируются.

Значение переменной можно изменить в любой точке программы, написав сначала имя переменной, затем оператор присваивания (=) и далее соответствующее выражение. Прежнее значение переменной будет заменено значением этого выражения. Например:

```
one=54;
two=one+three;
```

Слева от оператора присваивания должен находиться единственный объект или переменная. Следующее выражение недопустимо:

```
one+two=three;
```

Переменным типа float можно присваивать значения типа int, и наоборот. Когда значение с плавающей точкой присваивается целой переменной, то дробная часть значения с плавающей точкой отбрасывается (именно отбрасывается, а не округляется). Когда целое значение присваивается переменной с плавающей точкой, то предполагается, что дробная часть числа нулевая.

Выражения

Выражениями (expressions) называют составленные по определенным правилам последовательности идентификаторов переменных, арифметических операторов, вызовов функций, скобок и констант (чисел). Выражения показывают, как обрабатывать переменные и константы для получения результата выражения.

Арифметические операторы могут быть следующими:

+ сложение

- вычитание

* умножение

/ деление

% остаток от деления

Кроме того, в арифметических выражениях может использоваться оператор присваивания.

Определения

- Операторами выражения могут быть +, –, *, /, % и =.
- Членами выражения могут быть переменные (объекты), константы и вызовы функций.

Правила

- Соседние члены выражения должны быть отделены друг от друга оператором.
  - Два соседних члена выражения взаимодействуют согласно правилу, предписанному оператором.
- Операторы выполняются в соответствии с их приоритетом. Наибольшим приоритетом обладают операторы *, / и %, а наименьшим операторы + и -. Операторы, обладающие одинаковым приоритетом, выполняются в порядке их следования (слева направо).
- Перед членом выражения может стоять оператор –.
  - При этом значение становится отрицательным. (Перед членом выражения можно, хотя и не нужно, ставить оператор +.)

- Приоритет можно изменить с помощью скобок в любом месте выражения.
  - Выражения внутри скобок выполняются в первую очередь. Если имеется несколько пар скобок, то первой выполняется выражение внутри той пары, которая находится внутри всех остальных.
- Операторы инкремента (++) и декремента (--) могут указываться непосредственно до или после переменной.
  - Переменная при этом будетинкрементирована или декрементирована.
- Переменная (или объект) могут указываться в левой части оператора присваивания (=).
  - При этом результат выполнения выражения присваивается переменной, идентификатор которой указан слева от оператора присваивания.
  - В выражении может быть несколько операторов присваивания, но при условии, что перед каждым из них находится только одна переменная.



ПРИМЕЧАНИЕ Значения присваиваются переменным справа налево. Так, выражение howmany=times=times+1 является допустимым. Здесь сначала будет вычислено значение выражения times+1, потом оно будет присвоено переменной times, а затем— переменной howmany. Недопустимым является выражение howmany=times+1=5, поскольку выражение times+1, стоящее слева перед оператором присваивания, не состоит из одной переменной.

Самостоятельная практика

- Напишите функцию semmetry (athlete he, athlete she), в которой два гимнаста должны выполнять упражнения синхронно и симметрично.
  - Объект he должен выполнить следующую последовательность упражнений:

    Готов

    Руки вверх

    Руки влево

    Руки вверх

    Руки вправо

    Руки вверх

- Объект she должен выполнить следующую последовательность упражнений:

  Готов

  Руки вверх

  Руки вправо

  Руки вверх

  Руки влево

  Руки вверх

О Напишите программу, заставляющую гимнастов шесть раз выполнить эту последовательность упражнений.

Что нового мы узнали?

В этом уроке мы научились

0 Возвращать значения из функций.

0 Пользоваться встраиваемыми функциями.

0 Решать проблемы с помощью функций.

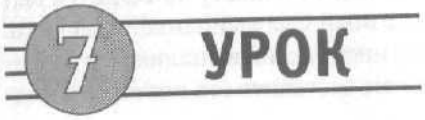
Часть III

Циклы

Компьютеры — это непревзойденное средство выполнения рутинной работы, а поскольку большая часть проблем связана с обработкой повторяющихся однотипных вычислений, ценность компьютеров еще более возрастает.

В этой части книги вы узнаете, как несколько раз выполнить тот или иной фрагмент программного кода, как задать количество таких повторений и, наконец, как получать результаты на каждом этапе выполнения.

Такие фрагменты программного кода называют циклами. Здесь вы познакомитесь с функционированием циклов и инструкциями, которые служат для их формирования. Вы научитесь использовать циклы для решения проблем и наконец начнете работу над своим первым настоящим приложением — упрощенным торговым терминалом.



Повторяющиеся вычисления

- Цикл while
- Цикл do...while
- Цикл for
- Условные циклы
- Вложенные циклы

В реальной гимнастической группе каждое упражнение гимнасту приходится выполнять многократно. Пусть, например, наш знакомый Сэл выполняет подскоки. Это простейший тип цикла — один и тот же набор инструкций выполняется заданное число раз. Если вы инструктор Сэла, то можете заставить его повторить подскок 15 раз.

Смышленый малый Сэл быстро сообразит, что делать. Он понимает значение слова *повторить*, а также знает, *что* и *сколько* нужно повторять.

К сожалению, компьютер не столь сообразителен, и ему надо объяснить все максимально подробно. Это легко сделать, если представить себе реального человека, который должен 15 раз выполнить упражнение.

Если вы собираетесь повторять какое-то действие, то должны подсчитывать, сколько раз оно уже выполнено, и остановиться, достигнув заданного значения. Это очень похоже на то, как если бы после каждого упражнения вы выкрикивали: «Раз! Два! Три!..». Этот процесс показан на рис. 7,1.

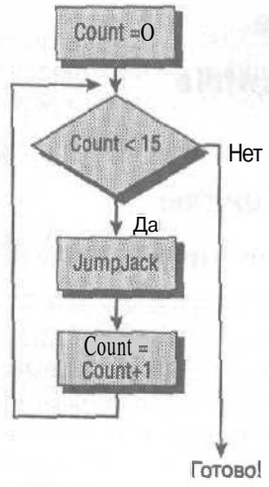


Рис. 7.1. Считаем и повторяем

Так же как и люди, компьютер должен считать, сколько раз он выполнил то или иное действие. Для этого используется целая переменная (в данном случае count). На этом уроке вы узнаете, как с помощью целой переменной задавать в ваших программах количество повторений.

Простые циклы

Сейчас мы рассмотрим простые *циклы* (loops), в которых тот или иной фрагмент программного кода выполняется заданное число раз. Таких циклов три:

- Цикл while
- Цикл do...while
- Цикл for

Цикл while

Показанный на рис. 7.1 цикл можно описать следующим образом:

Установка счетчика (переменная count) на ноль

Пока значение счетчика меньше 15, делать следующее:

Подскок (функция JumpJack())

Инкремент счетчика

Готово!

Здесь *счетчик* — это просто переменная, в которой хранится число выполненных подскоков. Вначале вы делаете значение счетчика равным нулю, поскольку еще ни одного подскока не выполнено.

Следующая строка, *Пока...*, указывает на то, что идущая далее (с отступом вправо) последовательность инструкций должна выполняться до тех пор, пока условие *значение счетчика меньше 15* будет оставаться истинным.

Почему последовательность инструкций сдвинута вправо? Об этом вы узнаете чуть позже.

Конечные циклы

На рис. 7.1 показана заданная нами последовательность выполнения инструкций. Это так называемая *блок-схема* (flowchart). Если начать с ее вершины (блок count=0) и следовать по стрелке, то следующим выполняемым блоком будет сравнение значения счетчика с числом 15 (блок count<15). В соответствии с результатом этого сравнения возможны два варианта. Однако, поскольку начальное значение счетчика равно нулю, мы переходим к выполнению упражнения (блок JumpJack) и инкременту счетчика (блок count=count+1).

После выполнения этих действий мы по стрелке возвращаемся к блоку сравнения значения счетчика с числом 15. Теперь это значение равно 1, что по-прежнему меньше 15, и вся последовательность инструкций повторяется снова. Таким образом, несколько блоков блок-схемы, которые нам приходится проходить снова и

снова, образуют замкнутую саму на себя последовательность инструкций. Именно по этой причине программисты называют многократно выполняемый фрагмент программного кода *циклом*.

В рассмотренном примере цикл выполняется 15 раз (значение счетчика меняется от 0 до 14 включительно). Когда показания счетчика достигают значения 14, программа выполняет функцию `JumpJack()` и затем значение счетчика увеличивается до 15. Теперь, когда значение счетчика сравнивается с числом 15 (блок `count<15`), программа выбирает альтернативный путь и мы покидаем цикл.

Бесконечные циклы

Предположим, мы забыли включить в программу следующую инструкцию:

```
Count=Count+1;
```

Когда в этом случае мы выйдем из цикла? Никогда! Мы будем все время сравнивать показания счетчика с числом 15. Каждый раз счетчик будет меньше 15 (поскольку его начальное значение не меняется). Результатом будет хорошо известный программистам «жучок», который называется *бесконечным циклом* (infinite loop). Этот пример предельно прост, но суть проблемы, я надеюсь, вам понятна. В других случаях причина возникновения бесконечного цикла может быть значительно более сложной и трудно уловимой.

Я уверен, вам еще не раз придется сталкиваться с бесконечными циклами, произведенными собственноручно (конечно, непреднамеренно).



СОВЕТ Если вам кажется, что выполняемая программа «попала» в бесконечный цикл, вы можете прервать ее работу, одновременно нажав клавиши Ctrl+Alt+Delete. Если вы работаете в среде Windows, то в результате на экране появится диалоговое окно, предлагающее вам прервать выполнение программы. После нажатия клавиши Enter выполнение программы завершится, и вы вернетесь в компилятор C++.

Составные инструкции

Посмотрим на алгоритм нашей программы с другой стороны. Зачем нужен отступ вправо в следующих строках:

Подскок (функция JumpJack())

Инкремент счетчика

Необходимо точно обозначать те инструкции, выполнение которых будет повторяться. В реальной программе, когда вы хотите, чтобы некоторая группа инструкций трактовалась как одна большая инструкция, она заключается в фигурные скобки и называется составной.

Составной инструкцией (compound statement) называется группа инструкций (заключенная в фигурные скобки), которая трактуется как одна инструкция. Такие составные инструкции (часто их называют *блоками)* используются в различных ситуациях и, в частности, в циклах. В алгоритмах эти группы инструкций сдвигают вправо, что в программах на C++, как вы должны помнить, означает фигурные скобки.



ПРИМЕЧАНИЕ Все функции, включая основную функцию mainprog(), представляют собой составные инструкции (именно поэтому после заголовка функции точка с запятой не ставится).



ВНИМАНИЕ Не ставьте точку с запятой между инструкцией while и составной инструкцией! Точка с запятой — это признак завершения инструкции.

Например, точка с запятой в следующем фрагменте кода заставляет компьютер думать, что пока переменная Count меньше 15, вы хотите повторить... *ничего:*

```
while(Count<15);   // Точка с запятой расположена неправильно!
{
   JumpJack();
   Count=Count+1;
}
```

Не напоминает ли эта конструкция бесконечный цикл? На первый взгляд, кажется, нет, но на самом деле это так! Такой пустой цикл будет повторяться вечно, поскольку значение переменной Count никогда не будет инкрементировано.

В правильно составленной программе после выполнения каждого цикла значение счетчика увеличивается. Это означает, что он последовательно принимает значения 1, 2, 3,..., 14. Когда значение счетчика становится равным 14, наше условие продолжает выполняться (14 меньше 15). Когда же мы повторим последовательность еще один раз (пятнадцатый), значение счетчика достигнет 15.

Теперь перед следующим повторением мы опять проверяем условие. Но поскольку счетчик (показывающий число сделанных упражнений) теперь равен 15, то условие оказывается ложным (15 не меньше 15) и повторения прекращаются! В результате цикл оказывается выполненным и программа продолжается, начиная со следующей инструкции (после закрывающей фигурной скобки).



ПРИМЕЧАНИЕ Можно инициализировать счетчик значением 1 (count=1) и проверять истинность выражения count<=15 или count<16, результат будет тем же.

На языке C++ наш алгоритм будет выглядеть следующим образом:

```
count=0;                // Установка счетчика в ноль
while(count<15)         // Повторение представленной ниже
                        // последовательности инструкций 15 раз
{
   JumpJack(Sal);       // Подскок
   count++;             // Инкремент счетчика
}                       // Конец последовательности инструкций
```

Таким образом, на рис. 7.1 была проиллюстрирована блок-схема цикла while.

Предыдущую программу можно представить более компактно:

```
count=0;
while(count<15)
{JumpJack(Sal);
count++;}
```

Однако согласитесь, что отступы делают программу более читабельной.

Операторы в цикле while

Ключевое слово while указывает, что последовательность инструкций должна выполняться до тех пор, пока условие истинно. Это условие заключено в скобки и отражает отношения между двумя значениями. Нас интересует случай, когда значение счетчика меньше 15, поэтому мы используем оператор < (меньше). Существуют и другие операторы:

== равно

> больше

>= больше или равно

< меньше

<= меньше или равно

!= не равно

Условия

Когда вы собираетесь повторить последовательность инструкций, то должны указать начало и конец цикла. Для этого нужно определить условие окончания цикла. Условие обычно представляет собой условное выражение, которое располагается непосредственно после ключевого слова while и заключается в скобки.

В нашем примере условным выражением является следующий фрагмент кода:

```
count<15
```

Оно означает, что мы собираемся повторять последовательность до тех пор, пока значение счетчика остается меньше 15. Символ < (меньше) называют *оператором отношения* (relational operator) или *условным оператором* (conditional operator), поскольку он отражает отношение между объектами. Конечно, вы можете использовать и другие операторы отношения. В некоторых случаях приходится задавать гораздо более сложные условия. На уроке 10 мы научимся составлять более сложные условные выражения.

Самостоятельная практика

- Составьте программу, в которой Сэл должен выполнить пять подскоков и пять поворотов вправо-влево.
- Составьте программу, в которой Сэл должен пять раз выполнить упражнение, состоящее из подскока и поворотов вправо-влево.
- Составьте программу, в которой сначала Сэл, а затем Салли должны выполнить по пять подскоков.
- Составьте программу, в которой Сэл и Салли должны одновременно выполнить по пять подскоков.
- Напишите функцию JmpJack() с двумя параметрами:

```
void JmpJack(athlete somebody, int howmany)
```

  - В этой функции объект somebody класса athlete должен повторить подскок столько раз, сколько задано целой переменной howmany.

Цикл do...while

Цикл do...while — это несколько измененный вариант цикла while. В цикле while условие проверяется перед началом цикла. Таким образом, если условие изначально не истинно, цикл вообще не будет выполнен!

В цикле do...while сравнение выполняется в конце цикла, как показано на рис. 7.2. Поэтому, вне зависимости от результата сравнения, один раз цикл всегда будет выполнен. Условие окончания цикла проверяется только после его первого выполнения.

В цикле do...while ключевое слово do указывает на начало цикла, а ключевое слово while, за которым следует условие, — на конец. Обратите внимание, что после условия в этом цикле ставится точка с запятой!

Например, свое упражнение Сэл может выполнить с помощью следующей программы:

```
count=0
do
```

продолжение

```
{
JmpJack(Sal);
count++;
}
while(count<15);
```

Обратите внимание, что сам цикл заключен в скобки, а инструкция while расположена после скобок.

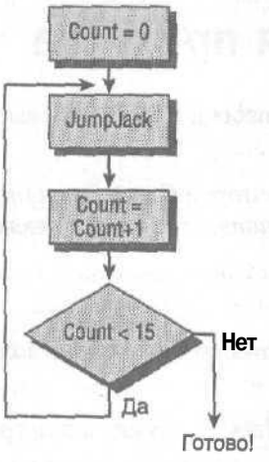


Рис. *7.2. Блок-схема цикла do...while*

Цикл for

Часто вместо цикла white удобнее использовать цикл for, поскольку для цикла while требуется инициализация счетчика и постоянное обновление его значений, и если случайно забыть то либо другое, цикл работать не будет.

Вспомним программу с циклом while:

```
count=0;
while(count<15)
{
   JmpJack(Sal);
   count++;
}
```

Эту программу можно записать иначе:

```
for(count=0; count<15; count++)
{
   JmpJack(Sal);
}
```

По своей структуре обе программы очень похожи, за исключением того, что у цикла for инициализация, проверка условия и инкремент счетчика находятся в одном месте, а именно внутри скобок, где эти три составляющие разделены точкой с запятой:

- Первая составляющая (count=0) представляет собой выражение, которое выполняется только однажды перед началом повторений. Очевидно, что это очень удобно для установки начального значения счетчика.
- Вторая составляющая (count<15) задает условие продолжения повторений. Как и в случае цикла while, это условие проверяется и повторение происходит только при его истинности.
- Третья составляющая (count++) представляет собой выражение, которое выполняется в конце каждого повторения. Очевидно, что это очень удобно для обновления значения счетчика.

Вместо предыдущей инструкции и с тем же успехом можно использовать следующую инструкцию:

```
for(count=1; count<=15; count++)
```

Иногда такая форма, при которой цикл начинается с единицы и заканчивается требуемым числом повторений, выглядит более естественно, чем цикл, начинающийся с нулевого значения.



ПРИМЕЧАНИЕ Помните, что выражения в инструкции с циклом for отделяются друг от друга точкой с запятой!

Внутри инструкции for можно объявлять новую переменную. Например:

```
for(int counter=1; counter<=15; counter++);
```

Один из вариантов использования цикла for показан в представленной ниже программе *c3askfor.cpp*. При этом число повторений вводится с клавиатуры посредством функции ask().

```
// Использование пользовательского ввода        //c3askfor.cpp
// Дата последней доработки 07.09.1994
#include "franca.h"
void JmpJack(athlete she)
{
   // Вызов функции JumpJack() объектом she
   she.up();
   she.ready();
}
```

```
void mainprog()
{
   athleteKim;
   int howmany, done;
   // Запрос: "Сколько раз нужно выполнить упражнение?"
   howmany=Kim.ask("How many jumping Jacks?");
   // Повторение цикла заданное количество раз
   for(done=1; done<=howmany; done++)
   {
      JumpJack(Kim);       // Вызов функции JumpJack()
      Kim.say(done);
   }
   Kim.say("done");        // Видите кавычки?
}
```

В рассмотренном примере мы заранее не знаем, сколько раз будет выполнена функция JmpJack(). Мы просто резервируем специальную переменную howmany и требуем от компьютера выполнять цикл до тех пор, пока число повторений (значение переменной done) меньше или равно значению переменной howmany.

Эквивалентность циклов for и while

Программный код цикла for полностью эквивалентен программному коду цикла while. Начальное выражение цикла for выполняется непосредственно перед инструкцией while, а последнее выражение цикла for — в самом конце цикла while.

Таким образом, последовательность инструкций:

```
for(<выражение1>; <выражение2>; <выражение3>)
{
   <инструкции>
}
```

Это абсолютно то же самое, что и следующая последовательность:

```
<выражение1>;
while(<выражение2>)
{
   <инструкции>
   <выражение3>;
}
```

Условия в циклах

В предыдущем разделе мы изучили простейшие типы циклов, в которых число повторений известно заранее.

Теперь мы займемся более сложным случаем, когда число повторений изначально не задано. Вы наверняка можете привести немало примеров из своей жизни, когда вам приходилось повторять что-то неопределенное число раз, пока не удавалось добиться выполнения того или иного условия.

Единственное отличие от обычных циклов состоит в методике проверки условия. Ранее мы следили за тем> чтобы значение счетчика оставалось меньшим либо равным заданного числа повторений. Теперь мы попытаемся следить за тем, что уже сделано.

Проверка вводимых значений

Интересное и полезное приложение циклов состоит в возможности проверки вводимых данных. Допустим, вы хотите, чтобы при вводе некоторого значения можно было удостовериться, что это значение не отрицательно. Этого можно добиться с помощью следующей последовательности инструкций:

```
float value;
// Запрос на ввод положительного числа
value=ask("Enter a positive value:");
while(value<0)
{
   // Ввод неверный.  Повторный запрос на ввод положительного числа
   value=ask("Wrong, enter a POSITIVE value:");
}
```

В этом фрагменте программы при каждом вводе отрицательного числа пользователю выдается сообщение с требованием повторного ввода. Такой способ проверки значений очень важен, и вам тоже имеет смысл почаще обеспечивать свои программы мерами предосторожности подобного рода. Никогда не следует доверять тому, что делает пользователь, но всегда можно доверять тому, что делает ваша программа!

Пример покупки

Допустим, нам нужно рассчитать сдачу за купленные в магазине продукты. Как это сделать, иллюстрирует программа c3store1.cpp.

```
#include "franca.h"
void mainprog()          //c3store1.cpp
{
```

продолжение

```
   float price, change, amount, tax=8.25;
   // Запрос на ввод цены товара
   price=ask("Input the price:");
   price=price+price*tax/100;
   Cout<<price;
   // Запрос на ввод уплаченной суммы
   amount=ask("Enter the amount:");
   while(amount<price)
   {
      // Запрос о необходимости доплаты
      amount=ask("Not enough, re-enter the amount:");
   }
   change=amount-price;
   Cout<<change;
}
```

После ввода уплаченной суммы (переменная amount) цикл while будет повторяться до тех пор, пока эта сумма меньше цены. Если сумма окажется достаточной, то цикл выполнен не будет (ни разу) и мы перейдем к расчету сдачи. С другой стороны, если уплаченной суммы окажется недостаточно, цикл обеспечит запрос на ввод ее нового значения. Таким образом, пока не будет введено правильного значения — неважно, сколько раз пользователь при этом ошибется, — выполнение программы продолжено не будет!

Того же самого можно добиться с помощью цикла do...while. Поскольку в этом случае условие проверяется после однократного выполнения цикла, в программе может быть одним вводом меньше. В рассмотренном примере заменим цикл while на цикл do...while:

```
do
{
   // Запрос на ввод уплаченной суммы
   amount=ask("Enter the amount:");
}
while(amount<price);
```

Обратите внимание, что цикл do...while в данном случае удобнее — строк стало меньше. Но с другой стороны, если пользователь постоянно вводит недостаточную сумму, программа никак его об этом не извещает.

Контроль времени

Помимо числа выполненных повторений имеется несколько других способов определения момента завершения цикла. Как уже упоминалось, одним из них явля-

ется проверка правильности вводимого значения. Другой интересный пример возникает, когда продолжительность цикла определяется по времени.

Как вы наверное уже догадались, для этого нужно написать правильные условия завершения цикла while, do...while или for.

Пример гимнастического класса

В качестве примера цикла с условием, проверяемым по времени, допустим, что некий гимнаст собирается выполнять упражнения в течение 12 секунд. Как вы понимаете, нельзя заранее сказать, сколько упражнений будет выполнено за 12 секунд (если, конечно, не знать продолжительность каждого упражнения). Эта ситуация вполне обычна для реальной жизни. В течение определенного интервала времени может происходить множество тех или иных событий, как это и показано в приведенном ниже примере программы c3dotime.cpp:

```
// Проверка момента окончания цикла по времени ... c3dotime.cpp
// Дата последней модернизации 08.08.1994
#include "franca.h"
void JumpJack(athlete she)  // Функция JumpJack()
{
   she ready();
   she up();
   she ready();
}
void mainprog()             // Основная программа
{
   athlete  Kim;            // Объявление объекта Kim класса athlete
   int  howmany=1;          // Объявление переменной для счетчика
   Clock timex;             // Создание часов
   while(timex.time()<12)   // Выполнение цикла до тех пор,
                            // пока не истекут 12 секунд
   {
      JumpJack(Kim);        // Выполнение функции JumpJack()
      Kim.say(howmany++);   // Вывод на экран числа повторений
   }
}
```

Выполните эту программу. Обратите внимание, что в условии цикла while время контролируется по часам (объект timex типа Clock) и цикл продолжается, пока оно меньше 12.

Пример определения быстродействия компьютера

Было бы интересно узнать, насколько быстр объект Kim на вашем компьютере. Если вы модифицируете предыдущую программу, разместив ноль в качестве аргументов сообщений ready и up, то эти движения будут выполняться максимально быстро. Проверьте, например, сколько упражнений можно выполнить за 12 секунд. Результат будет зависеть от типа вашего компьютера. Вы можете, таким образом, сравнить свой компьютер с компьютерами своих коллег. Чем больше упражнений будет выполнено, тем быстрее соответствующий компьютер.

Функция yesno()

В некоторых ситуациях вам может понадобиться, чтобы программа запрашивала пользователя о необходимости продолжать цикл. В качестве примера вернемся к программе c3store1.cpp. В этой программе мы рассчитывали сдачу на основе введенных значений цены товара и уплаченной суммы.

Вполне вероятно, что в течение дня эта процедура будет многократно повторяться, а нам бы не очень хотелось снова и снова запускать нашу программу, когда покупателю нужно сделать покупку. В этом случае оптимальным решением будет заставить программу работать непрерывно. Но как в этом случае ее остановить? Мы можем в конце каждого цикла выводить на экран запрос: *Do you want to continue? Yes or No?* (Хотите продолжать? Да или нет?). Если пользователь отвечает *Yes,* цикл повторяется.

А как узнать ответ пользователя?

Нам на помощь приходит заголовочный файл franca.h, в который включена функция, отображающая диалоговое окно с кнопками Yes и No. В качестве аргумента для этой функции можно задать некую фразу (в данном случае это фраза *Do you want to continue ?*),которая тоже появится на экране. На рис. 7.3 показано диалоговое окно, которое создает функция yesnoQ.

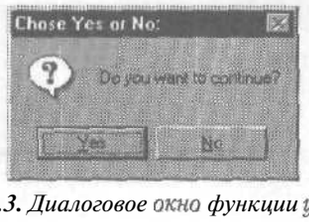


***Рис. 7.3.** Диалоговое окно функции yesno()*

Такое диалоговое окно появляется после следующего вызова функции yesno():

```
yesno("Do you want to continue?");
```

Функция yesno() также возвращает значение, что позволяет использовать вызов функции в выражении. Если пользователь нажимает кнопку Yes, возвращаемое

значение будет равно единице, а если No — нулю. Ниже представлен переделанный вариант программы, который назван c3store2.cpp.

Обратите внимание, что мы включили тело предыдущей программы внутрь цикла do...while. Функция yesno() вызывается непосредственно в выражении проверки окончания цикла.

```
#include "franca.h"
void mainprog()            //c3store2.cpp
{
   float price, change, amount, tax=8.25;
   do
   {
   // Запрос на ввод цены товара
   price=ask("Input the price:");
   price=price+price*tax/100;
   Count<<price;
   // Запрос на ввод уплаченной суммы
   amount=ask("Enter the amount:");
   while(amount<price)
   {
      // Запрос о необходимости доплаты
      amount=ask("Not enough, re-enter the amount:");
   }
   change=amount-price;
   Count<<change;
   }
   // Запрос о необходимости продолжать работу
   while(yesno("Do you want to continue?"));
}
```

Функцию yesno() можно использовать не только в циклах do...while, но и в других выражениях.



СОВЕТ Поскольку ответ *Yes* задан по умолчанию, формируйте свой вопрос так, чтобы ответом на него в большинстве случаев было *Yes*. Тогда пользователю можно будет сразу нажимать на клавишу Enter и не перемещать указатель мыши, чтобы щелкнуть на кнопке No.



ПРИМЕЧАНИЕ Функция yesno() не относится к стандартным функциям языка C++ и доступна только в программном обеспечении книги.

Вложенные циклы

Часто возникают ситуации, при которых группу повторяющихся команд приходится вкладывать внутрь другой группы повторяющихся команд. Например, пусть нам надо принимать по две таблетки в день в течение 10 дней. Мы 10 раз повторяем действие, которое само является повторением. Таким образом, мы получаем следующий алгоритм:

Повторить следующую инструкцию 10 раз:

    Повторить следующую инструкцию 2 раза:

        Принять таблетку

        Подождать следующего дня

В результате вы примете все 20 таблеток.

Обратите внимание, что здесь одно повторение вложено внутри другого. Чтобы перейти к внешнему повторению, нужно выполнить все внутренние. На рис. 7.4 показан *внутренний* (inner loop), или *вложенный, цикл* (nested loop), который повторяется каждый день, а на рис. 7.5 показан полный процесс, в который входят как *внешний* (outer loop), так и *внутренний* циклы.



Рис. 7.4. *Внутренний цикл, повторяемый каждый день*

Другой пример вложенных циклов можно найти в гимнастическом классе. Предположим, что инструктор предлагает вам сделать следующее:

Повторить следующую инструкцию 5 раз:

    Повторить следующую инструкцию 10 раз:

        Подскок

    Повторить следующую инструкцию 5 раз:

        Влево-вправо

Повторить следующую инструкцию 3 раза:

    Подскок



Рис. *7.5. Полный десятидневный цикл*

Вы должны сделать 10 подскоков, потом пять разворотов влево-вправо, повторить всю последовательность *5* раз (в итоге получится 50 подскоков и 25 разворотов) и, наконец сделать 3 дополнительных подскока.

Первая последовательность подскоков, в отличие от второй, вложена внутрь другого цикла. Соответствующая программа на C++ имеет следующий вид:

```
#include "franca.h"
void mainprog()
{
   athlete Sal;
   int bigcount, smallcount;                    // Переменные-счетчики
   for(bigcount=1; bigcount<=5; bigcount++)   // Внешний цикл
   {
      for(smallcount=1; smallcount<=10; smallcount++)
      {
         JmpJack(Sal);
      }
      for(smallcount=1; smallcount<=5; smallcount++)
      {
```

продолжение

```
        leftright(Sal);
      }
    }                                  // Конец внешнего цикла
  for(smallcount=1; smallcount<=3; smallcount++)
  {
    JmpJack(Sal);
  }
}
```

Циклы и робот Tracer

Теперь, познакомившись с циклами, мы можем по-новому направлять движения робота. Например, теперь гораздо проще составить функцию для рисования линии любого заданного размера. Для этого необходимо просто включить в нее параметр, определяющий число шагов. С помощью другого параметра желательно сохранить возможность выбора цвета.

Заголовок этой функции должен выглядеть примерно так:

```
void Line(int size, intcolor=2)
```

Функция должна заданное число раз повторять уже знакомую нам последовательность инструкций:

```
mark(color);
step();
```

Если параметр задания длины линии представляет число шагов, то цикл будет повторяться на один раз меньше (вспомните наш первый квадрат).

Проще всего это сделать с помощью цикла for:

```
for(int howmany=1; howmany<size; howmany++)
```

Благодаря условию howmany<size и начальному значению howmany=1 цикл будет выполняться size-1 раз.

Полностью функция имеет следующий вид:

```
void Line(int size, int color=2)
{
  for(int howmany=1; howmany<size; howmany++)
  {
    Tracer.mark(color);
    Tracer.step();
  }
}
```

Перед тем как двигаться дальше, отметьте, что теперь для изменения числа шагов не нужно заново переписывать функцию. Для этого достаточно изменить значение параметра. Ниже показана программа c3square.cpp, предназначенная для рисования двух квадратов разных цветов и размеров.

```
#include "franca.h"    //c3square.cpp
Robot Tracer;
// Цикл делает функцию line проще
void line(int size, int color=2)
{
   for (int howmany=1; howmany<size; howmany++)
   {
      Tracer.mark(color);
      Tracer.step();
   }
}
// Цикл делает функцию square() проще
void square(int size, int color=2)
{
   for(int turn=1; turn<=4; turn++)
   {
      line(size, color);
      Tracer.right();
   }
}
// Основная функция mainprogO осталась прежней
void mainprogO
{
   Tracer.face(3);
   square(3);
   Tracer.step(3);
   square(4,5);
}
```

Исследование комнаты

Как научить робота обходить стены прямоугольной комнаты и возвращаться обратно?

Следует воспользоваться тем, что с помощью следующего сообщения робот может определять наличие преграды перед собой:

```
Tracer.seewall();
```

Еще лучше заставить его сохранять направление движения, пока истинно условие:

```
Tracer.seenowall();
```

Это означает, что мы предоставляем возможность роботу двигаться, пока он не наткнется на стену, При этом количество необходимых шагов заранее знать не нужно. До тех пор пока условие Tracer.seenowall() остается истинным, робот будет продолжать движение. Для этого нужна следующая программа:

```
for(; Tracer.seenowall(); )
{
   // Можно еще и помечать пройденный путь
   Tracer.mark();
   Tracer.step();
}
```

Что случилось с двумя условиями внутри инструкции for? Они исчезли! В данном случае они не нужны. Не нужно ни инициализации, ни каких-либо действий после каждой итерации.

Если вам необходимо сосчитать количество пройденных роботом шагов, пока он не уперся в стену, код можно несколько изменить:

```
for(int howmany=0; Tracer.seenowall; howmany++)
{
   // Можно еще и помечать пройденный путь
   Tracer.mark();
   Tracer.step();
}
```

Решение проблемы

Допустим, у нас имеется пустая прямоугольная комната (как показано на рис. 7.6), в углу которой, повернувшись на восток, стоит робот (объект Tracer).

Можете сами догадаться, как заставить робота обойти комнату? Поскольку вы знаете, что комната прямоугольна, можно заставить робота сделать следующее:

- Шагать вперед до тех пор, пока не встретится стена. После того как встретится первая стена, останутся еще три,
- Повернуть и идти до следующей стены. После этого останется только две! Что дальше?

Сможете закончить сами? Попробуйте! А потом сравните с представленными ниже решениями.

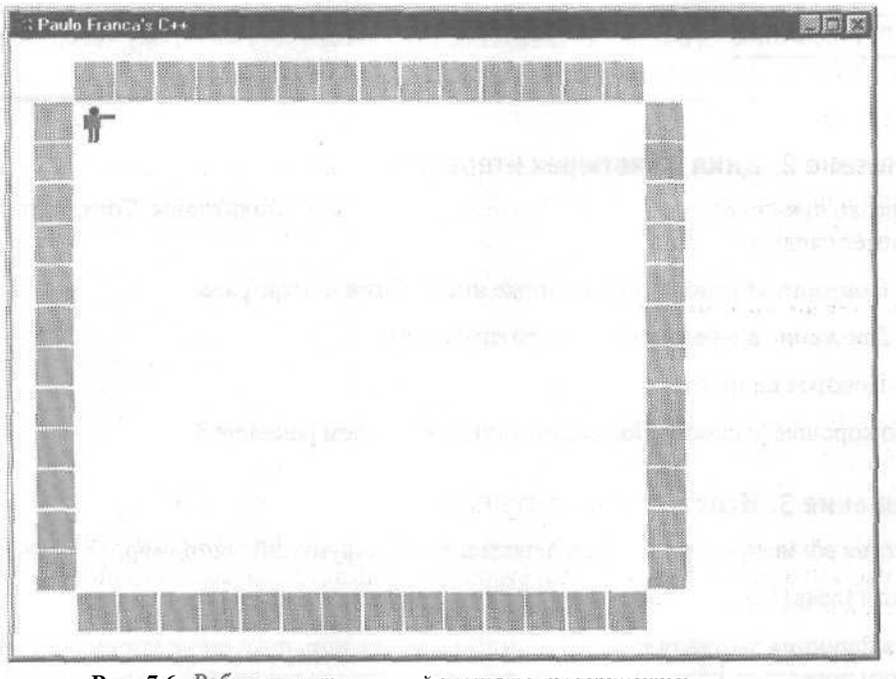


Рис. *7.6. Робот стоит в пустой комнате, повернувшись на восток*

Решение 1. Прогулка в четырех стенах

В этом решении робот двигается по очереди вдоль каждой из стен. Он идет до первой стены, поворачивает направо, идет до следующей, опять поворачивает направо и т. д.

Вот возможный алгоритм:

Движение вперед, пока не встретится стена

Поворот направо

Движение вперед, пока не встретится стена

Поворот направо

Движение вперед, пока не встретится стена

Поворот направо

Движение вперед, пока не встретится стена

Поворот направо

Первое решение правильно, хотя и не интересно. Одна и та же последовательность двух инструкций записывается целых четыре раза. Очевидно, что здесь мы имеем дело с циклом.



ПРИМЕЧАНИЕ Никогда не следует повторять одну и ту же строку несколько раз подряд!

Решение 2. Цикл из четырех итераций

Гораздо лучше создать цикл из четырех путешествий вдоль стены. Тогда алгоритм примет следующий вид:

Повторить представленные ниже инструкции четыре раза:

Движение вперед, пока не встретится стена

Поворот направо

Это хорошее решение. Возможно, оно не хуже, чем решение 3.

Решение 3. Использование функции

Третья возможность связана с использованием функций, например:

```
void findwall();
```

Эта функция заставляет робот двигаться до тех пор, пока он не упрется в стену, а затем повернуть направо. Если функция составлена правильно, то алгоритм будет выглядеть так:

Повторить следующую инструкцию четыре раза:

Вызов функции findwall()

Решение 4. Использование возвращаемого значения функции

Последним рассмотрим решение, связанное с использованием возвращаемого значения функции. Пусть функция возвращает число ячеек, которые робот прошел перед тем, как уперся в стену.

Пример такой функции:

```
int findwall()
{
   int steps=Q;
  while(Tracer.seenowall())
   {
      Tracer.step();
      step++;
   }
   tracer.right();
   return steps;
}
```

Преимущество этой функции в подсчете числа шагов. Если не сейчас, то со временем это может пригодиться. Например, в специальной переменной можно запоминать пройденное роботом число шагов:

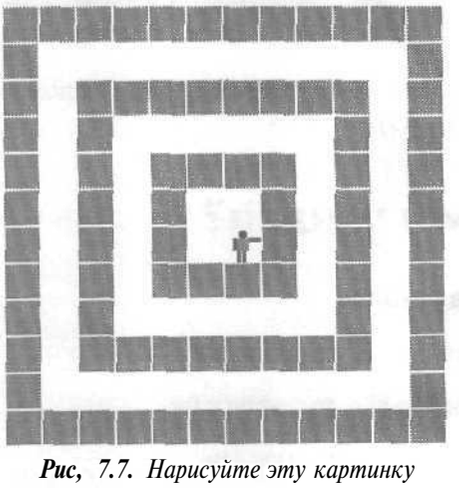
```
int howfar;
howfar=findwall();
```

Хотя эта функция и возвращает значение, нам оно пока ни к чему, поэтому использовать функцию можно так же, как ее предыдущую версию:

```
for(int=1; i<4; i++)
findwall();
```

Самостоятельная практика

- Напишите функцию int distance().
  - Эта функция призвана измерять число шагов, которое должен пройти робот, чтобы упереться в стену. После измерения расстояния робот должен вернуться обратно и развернуться в прежнем направлении.
- Предположим, что робот находится где-то внутри прямоугольной комнаты, но вы не знаете, где именно и в какую сторону он повернут. Заставьте его тем не менее измерить комнату.
- Предположим, что робот находится где-то внутри прямоугольной комнаты, но вы снова не знаете, где именно и в какую сторону он повернут. Заставьте его тем не менее обойти всю комнату.
- Нарисуйте картинку, показанную на рис. 7.7.



Рис, 7.7. *Нарисуйте эту картинку*

- Нарисуйте картинку, показанную на рис. 7.8.

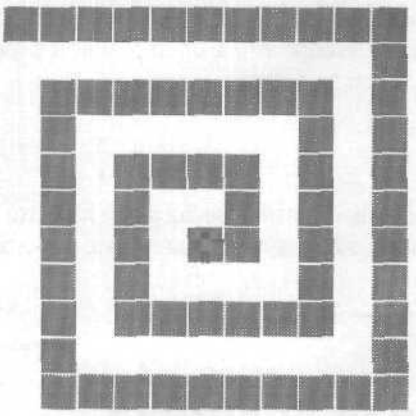


Рис. 7.8. Нарисуйте эту картинку

О Нарисуйте картинку, показанную на рис. 7.9.

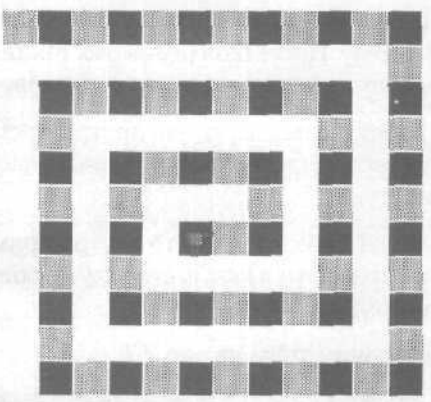


Рис. 73. Нарисуйте эту картинку

О Какие функции необходимо составить, чтобы с их помощью нарисовать эти картинки (см. рис. 7.7—7.9)?

Что нового мы узнали?

В этом уроке мы научились

0 Писать повторяющиеся фрагменты кода.

0 Использовать циклы while, do...while и for.

0 Задавать условия выполнения циклов.

0 Создавать вложенные циклы.



УРОК 8 Разработка базовых циклов

- **Циклы с** параметрами
- **Первый этап создания циклов**
- **Правила и** соглашения

Повторяющиеся в циклах фрагменты кода вовсе не обязательно должны быть одинаковы, В самом деле, в природе гармония достигается благодаря тому, что даже представители одного вида в чем-то обязательно отличаются друг от друга. Вы можете спросить, как повторяющиеся операции могут отличаться друг от друга, ведь тогда они уже не будут повторяющимися.

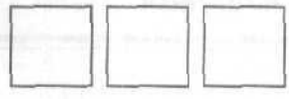
Но вы сами постоянно сталкиваетесь с этим! Каждое утро вы просыпаетесь, одеваетесь, завтракаете. И так изо дня в день. Но при этом:

- Вы не надеваете одну и ту же одежду.
- Вы не едите один и тот же завтрак.

Таким образом, хотя в общих чертах вы повторяете одни и те же действия, в деталях они различаются.

Повторение с изменением

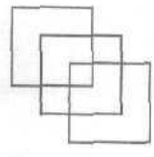
На рис. 8.1 и 8.2 показано несколько вариантов изображения трех квадратов.



Горизонтальный сдвиг



Еще один вид горизонтального сдвига



Горизонтальный и вертикальный сдвиг

Рис. 8.1. *Три квадрата одинакового размера*



Начальное положение



Горизонтальный сдвиг



Горизонтальный и вертикальный сдвиг

Рис. 8.2. *Три квадрата разного размера*

При наличии подходящих инструкций рисования квадрата вы, вероятно, отметили бы, что в каждом случае их нужно повторить трижды, но при этом очередной квадрат должен отличаться от предыдущего либо размером, либо положением, либо тем и другим.

Если вы умеете рисовать квадрат заданного размера в заданном месте, то сможете заставить робота нарисовать для вас любую из этих картинок. Пусть, например, самый большой квадрат имеет размер 12 ячеек, а остальные 8 и 4.

Я могу дать вам совет по поводу рисования одной из картинок:

- Обратите внимание на верхнюю картинку рис. 8.2. Все квадраты имеют одну вершину, но их размеры соответственно 12, 8 и 4! Другими словами, вы начинаете с размера 12 и каждый раз рисуете новый квадрат, уменьшая его размер на 4 ячейки.

Рассмотрим, как это делается.

Пример рисования квадратов разного размера

Сначала напишем алгоритм;

Рисование квадрата размером 12 ячеек

Рисование квадрата размером 8 ячеек

Рисование квадрата размером 4 ячейки

Сейчас вы уже знаете, как для рисования квадратов использовать функцию. Для этого вам не нужно ее создавать! У вас уже есть готовая функция, и вы можете просто включить ее в свою программу.

Первая программа

Наша первая попытка выглядит следующим образом:

```
#include "franca.h"
Robot Tracer;
void mainprog()
{
   square(12);
   square(8);
   square(4);
}
```

Это вполне работоспособная программа, но ее можно сделать лучше.



ПРИМЕЧАНИЕ Никогда не следует повторять одну н ту же строку несколько раз подряд!

Вы обратили внимание, что программа просто последовательно рисует квадраты? Причем трижды! Хороший программист всегда заметит повторяющиеся фрагменты программного кода, сколько бы их не было: 3, 15 или 100, и постарается упростить программу. Мы поступим так же.

Вторая программа (неправильная)

Наша вторая программа должна выглядеть примерно так:

```
#include "franca.h"
Robot Tracer;
void mainprog()
{
   int many, size;
   size=12;
   for(many=1; many<=3; many++)
   {
      square(size);
   }
}
```

Такая программа хотя и не имеет ошибок, но будет рисовать все квадраты одинакового размера. Размер квадрата на каждой итерации должен уменьшаться на четыре ячейки. Если вы знаете, как найти размер каждой следующей ячейки, то сможете подсчитать размер всех.

Третья программа

Наша третья программа выглядит следующим образом:

```
#include "franca.h"          //c3sq3.cpp
Robot Tracer;
void mainprog()
{
   int many, size;
   size=12;
   for(many=1; many<=3; many++)
   {
      square(size);
      size=size-4;
   }
}
```

Понимаете, что происходит с размером квадрата? Вначале он был равен 12 ячейкам, Вы начинаете цикл и рисуете первый квадрат этого размера. Далее вы вычисляете новый размер, просто вычитая 4 из его предыдущего значения. Таким образом, при следующей итерации вы будете рисовать квадрат размером 8 ячеек и т. д.

Чтобы нарисовать другие картинки из рис. 8.2, на которых квадраты сдвинуты относительно друг друга, вы должны каждый раз перемещать робота в новое исходное положение. На первый взгляд кажется, что все эти перемещения отличаются друг от друга. Совершенно не отличаются! Если внимательно посмотреть на картинки, то обнаруживается, что каждый следующий квадрат сдвинут относительно предыдущего на строго определенное число шагов по вертикали, по горизонтали или по обоим направлениям. Все предельно просто.

Далее мы соответствующим образом дополним программу *c3sq3.cpp*.

Преимущества циклов

Первая и третья программы рисуют одну и ту же картинку! Однако я надеюсь, что вам больше понравилась третья. Возможно, мой выбор вас удивит. Хотя разобраться легче, пожалуй, в первой программе, я все-таки настаиваю, что третья программа лучше. Почему?

Сравнивая эти программы, вы можете сказать, что первая попытка не менее хороша, чем последняя. Действительно, по своему объему программы не слишком разнятся, зато первая гораздо понятней. Действительно, пока мы рисуем всего три квадрата, первая программа кажется соблазнительно простой. Но что будет, если нам придется делать сто или тысячу повторений. Чтобы, например, нарисовать 10 квадратов, к первой программе придется добавить еще семь инструкций. А сколько инструкций нужно добавить к последней программе? Ни одной! Все, что нужно сделать, — это изменить исходный размер квадрата и число повторений.

Еще одним существенным преимуществом циклов является то, что число итераций может быть заранее неизвестно. Первая программа рисует три и только три квадрата. В последней же программе требуемое число квадратов можно задать в качестве параметра цикла. Если заданное число итераций хранить в переменной total, то соответствующий цикл будет выглядеть так:

```
for(many=1; many<=total; many++)
```

Организация циклов

Роль циклов в программе переоценить невозможно. Чем больше вы их создадите, тем лучше. Иногда бывает так трудно определить, чем один этап отличается от другого, что идентификация ситуаций для организации циклов становится весьма тонким делом. Со временем вы наберетесь достаточно опыта и вам станет легче находить соответствующие отличия.

Рисование трех квадратов можно выполнить также и с помощью вложенных циклов. У каждого квадрата по четыре стороны. Таким образом, три квадрата имеют 12 сторон, и мы можем составить программу со следующим алгоритмом:

Установка начального размера квадрата в 12 ячеек

Повторить следующие инструкции 3 раза:

Повторить следующие инструкции 4 раза:

Нарисовать линию заданного размера

Повернуть направо

Вычислить новый размер

Четвертая программа

Соответствующая программа на C++ будет выглядеть следующим образом:

```
#include "franca.h"
int size, squarecount, linecount;
size=12;
for(squarecount=1; squarecount<=3; squarecount++)
{
   for(linecount=1; linecount<=4; linecount++)
    {
       line(size);
       Tracer.right();
   }
   size=size-4;
}
```

Результат выполнения программы останется прежним. В предыдущей версии роль внутреннего цикла играла функция square(). Я думаю, вы согласитесь, что в результате третья программа была понятнее. Однако создать функции на все случаи жизни невозможно, поэтому иногда предпочтительней использовать вложенные циклы.

Самостоятельная практика

- Составьте программу для рисования квадратов средней картинки рис. 8.2.
- Составьте программу для рисования квадратов нижней картинки рис. 8.2.

Первый этап создания циклов

Широкое распространение компьютеров во многом связано с тем, что большинство реальных проблем можно описать в виде циклов. Процедура обслуживания покупателя одинакова и повторяется тысячи раз в день. Точно так же обработка сложных научных расчетов, как правило, подразумевает многократное выполнение одних и тех же фрагментов программного кода.



ПРИМЕЧАНИЕ Дополнительную информацию о разработке циклов можно получить в разделе «Второй этап создания циклов» урока 10.

Теперь вы знаете, как управлять многократным повторением одного фрагмента программного кода, и нужно научиться пользоваться этими знаниями. Бывают ситуации, в которых возможность организации цикла очевидна, но как реализовать эту возможность, совершенно непонятно.

Общие положения

Чтобы организовать цикл, необходимо четко определить следующее:

- Для решения какой задачи предполагается организация цикла?
- Какие изменения должны происходить после каждой итерации?
- Когда предполагается заканчивать итерации?
- Какие начальные условия организации цикла?

Циклические задачи

Я предлагаю прежде всего постараться определить, для решения какой задачи лучше использовать цикл. Возможно, сначала следует рассмотреть упрощенный слу-

чай, когда требуется всего одна итерация. Полезно также подумать, какая задача будет полностью решена после завершения каждой итерации. Попытайтесь сформулировать утверждение, которое остается истинным после завершения каждой итерации.



ПРИМЕЧАНИЕ Утверждение, остающееся истинным после завершения каждой итерации, называется *инвариантом цикла* (loop invariant).

Например, вспомним функцию рисования квадрата программы c3sq3.cpp:

```
square(12);
```

Такой вариант устроит нас только тогда, когда нам нужен всего один квадрат.

Однако размер квадратов каждый раз меняется. Следовательно, размер относится к тем атрибутам цикла, которые меняются после каждой итерации. С другой стороны, истинным для каждой итерации остается следующее утверждение:

Рисование квадрата номер *n*

Изменения при каждой итерации

Далее надо попытаться определить, что меняется при каждой итерации.

В предыдущем примере менялся только размер квадрата. Каждый следующий квадрат уменьшался на 4 ячейки. Такое изменение данных можно выполнить внутри самого цикла, а точнее в третьем выражении инструкции for. Вполне вероятно, что вам также потребуется вычислять количество выполненных итераций,

Завершение цикла

Убедитесь, что цикл тем или иным способом завершается (например, путем подсчета итераций). На уроке 10 вы узнаете, как прервать выполнение цикла путем проверки некоторого условия. В любом случае важно быть абсолютно уверенным, что событие, необходимое для завершения цикла, обязательно произойдет.

В данном примере мы собирались изобразить три квадрата. Следовательно, цикл остановится, когда значение счетчика станет равным трем.

Начало цикла

Убедитесь в правильности всех исходных значений, необходимых для начала цикла. Начальное значение счетчика должно быть равно нулю или единице. Установите также начальные значения остальных переменных.

Иногда проще задать начальные условия уже после создания цикла. В рассмотренном примере начальный размер квадрата равен 12, а значение счетчика числа квадратов равно 1.

Пример разработки цикла

Предположим, что преподаватель попросил вас рассчитать среднюю успеваемость студентов вашей группы. Как решать эту задачу?

Очевидно, нужно сложить все оценки и получившееся значение разделить на число студентов. Общая последовательность действий должна выглядеть так:

Сложить все баллы

Рассчитать средний балл

Написать средний балл

А как вы будете складывать баллы?

Как и при использовании калькулятора, их нужно ввести в память:

Повторять следующие инструкции,
пока остаются неучтенные студенты:

Ввести значение

Нажать клавишу + (плюс)

Поскольку в калькуляторе нажатие клавиши + (плюс) добавляет текущее значение к накапливаемой сумме, на компьютере это будет выглядеть аналогично:

Повторять следующие инструкции,
пока остаются неучтенные студенты:

Ввести значение

Прибавить введенное значение к общей сумме

Если вам трудно понять этот принцип расчета, можно попытаться написать процедуру вычисления среднего балла для двух или трех студентов, а затем преобразовать ее в цикл.

Теперь вы знаете те инструкции, которые требуют повторения:

Ввести значение

Прибавить введенное значение к общей сумме

На C++ эти инструкции выглядят следующим образом:

```
// Запрос на ввод очередного балла
grade=ask("Please enter next grade:");
total=total+grade;
```

Вы теперь можете проверить, какие изменения требуются при переходе от одной итерации к другой. При каждой итерации вводится значение очередного балла и при этом обновляется накапливаемая сумма. Больше никаких изменений нет.

Завершение цикла

Необходимо выбрать механизм завершения цикла. Проще всего использовать для этой цели общее число студентов. Например, если их было 26, подойдет цикл следующего типа:

```
for(int which=0; which<26; which++)
```

Будет такая программа работать? Несомненно. Но вот гибкости в ней нет. Через месяц преподаватель попросит вас написать новую программу, так как вполне вероятно, что в группе к тому времени останется всего 24 студента.

Здесь имеется несколько возможных вариантов:

- Можно выдавать запрос на ввод реального числа студентов.
- Можно выдавать запрос о необходимости продолжать выполнение очередной итерации.
- Можно определить специальное сигнальное значение, которое будет указывать на то, что ввод данных закончен.

Первый вариант самый простой. Просто включите несколько дополнительных инструкций перед началом цикла и слегка измените инструкцию for:

```
int class_size;
// Запрос на ввод числа студентов
class_size=ask("Enter the number of grades:");
for (int which=0; which<class_size; which++)
```

Это неплохое решение, но оно требует, чтобы кто-то заранее рассчитывал число студентов. Пользователю это может не понравиться, особенно если число студентов достаточно велико.

Благодаря функции yesno() второй вариант также легко осуществим. Помните, что у вас есть возможность узнать мнение пользователя с помощью кнопки Yes или No диалогового окна.

Воспользуйтесь функцией yesno() внутри следующего условия, где пользователю выдается запрос о необходимости введения очередного балла:

```
for(int which=0; yesno("Input another grade?"); which++)
```

Если пользователь нажмет на кнопку Yes, цикл будет продолжаться и запрашивать новые значения.

Поскольку в данном случае для завершения цикла число итераций подсчитывать не требуется, соответствующая переменная which для этой цели больше не нужна. Тем не менее в ней по-прежнему хранится число введенных значений, что в конечном итоге позволяет нам узнать общее число студентов в классе. Отметим, что если

использовать другую инструкцию, то к моменту завершения цикла значение счетчика which будет на единицу больше числа введенных данных:

```
for(int which=1; which<class_size; which++)
```

Таким образом, чтобы в дальнейшем правильно рассчитать среднее значение, из него нужно вычесть единицу!



ПРИМЕЧАНИЕ Этот вариант неудобен тем, что перед введением очередного значения пользователю приходится нажимать кнопку Yes или No.

Наконец, можно использовать специальное значение, сигнализирующее о завершении ввода данных. Для этого вместо ввода очередного балла пользователю потребуется ввести какое-нибудь заведомо неправильное значение, показывающее, что ввод данных закончен. Так, например, в нашем примере балл (оценка) не может быть отрицательным, поэтому в качестве такого сигнального значения подойдет отрицательное число.

О том, как правильно применять подобную технику, вы сможете узнать на уроке 10.

Начальные условия

После создания цикла проверьте правильность задания начальных условий. Например, исходное значение переменной total должно быть равно нулю.

Полностью программа расчета средней успеваемости студентов выглядит следующим образом:

```
include "franca.h"
void mainprog()            //c3avgrd.cpp
{
   float grade, total=0;
   float average;
   for(int which=1; yesno("Input another grade?"); which++)
   {
      // Запрос на ввод очередного балла
      grade=ask("Enter next grade: ");
      total=total+grade;
   }
   average=total/which;
   Count<<average;
}
```

Правила и соглашения

В этом разделе обобщены правила, о которых мы узнали из предыдущих уроков.

Составные инструкции

Группа инструкций, начинающаяся с открывающей фигурной скобки { и заканчивающаяся закрывающей фигурной скобкой }, трактуется как одна составная инструкция. Например:

```
for(int times=1; times<=10; times++)
// Следующие строки представляют собой составную инструкцию
{
   Kim.JumpJack();
   Kim.say(times);
}
```

Вы должны были заметить, что тело функции также является составной инструкцией. Например:

```
void mainprog()        // Следующие строки представляют собой
                       // составную инструкцию
{
   athlete Kim;
   Kim.up();
   Kim.ready();
}
```

Вложенные инструкции

В одну составную инструкцию может входить другая составная инструкция. Например:

```
voidmainprogC)
{
   athlete  Kim;
   for(int times=1; times<=10; times++)
   {
      Kim.JumpJack();
      Kim.say(times);
   }
}
```

Каждая закрывающая скобка свидетельствует об окончании составной инструкции. По этой причине после нее точка с запятой не ставится.



СОВЕТ Хотя с точки зрения синтаксиса это не обязательно, при написании составной инструкции лучше сдвигать ее строки вправо.

Область видимости

Любая переменная (или объект), объявленная внутри составной инструкции, будет создаваться внутри нее (при выполнении инструкции с соответствующим объявлением). После выполнения составной инструкции любая переменная (или объект), объявленная внутри составной инструкции, удаляется. Например:

```
int i, j;
for(i=0; i<3; i++)
{
   intm;
   m=i+3;
}
```

В рассмотренной последовательности переменная т объявлена внутри составной инструкции for. Эта переменная создается одновременно с началом цикла и доступна вплоть до его завершения (если цикл позднее будет перезапущен, соответствующая переменная будет создана снова). После завершения цикла значение переменной m получить невозможно. Любая ссылка на нее воспринимается как синтаксическая ошибка.



ПРИМЕЧАНИЕ Если объявление переменной m находится внутри цикла, это не означает, что переменная будет создаваться на каждой итерации. Она создается только один раз одновременно с началом цикла.

Простые условия

Условие определяется посредством *условного выражения* (conditional expression) или *выражения отношения* (relational expression). Как это понятно из названия, выражение отношения определяет отношение между переменными или объектами. С помощью выражения отношения два объекта сравниваются между собой, и выдается тот или иной результат.



ПРИМЕЧАНИЕ Условие всегда заключается в круглые скобки.

Результат, который мы привыкли считать истиной либо ложью, в C++ представлен целым числом. Ноль обозначает *ложь*, а единица — *истину*.

Равенство и присваивание

Особого внимания заслуживает сравнение, устанавливающее равенство двух значений. В этой ситуации весьма распространенной является следующая ошибка:

```
if(a=b) ...;
```

В C++ такое выражение приемлемо с точки зрения синтаксиса, но не имеет никакого отношения к сравнению значений a и b. На самом деле это выражение присваивает значение переменной b значению переменной a (при этом исходное значение переменной a теряется). При правильном сравнении должен использоваться оператор равно == (не путать с оператором присваивания =):

```
if(a==b) ...;
```



СОВЕТ Непреднамеренное использование выражения if(a=b) ... будет оттранслировано компилятором и приведет к неприятному и трудноуловимому «жучку» в вашей программе. Если значение переменной Ь ненулевое, то результат выполнения выражения будет *истинным*. Если же значение переменной b нулевое, — *ложным*. Старайтесь по возможности избегать выражений подобного рода.

Объединение арифметических и условных выражений

Хотя я не советую вам этого делать, но вполне допустимо объединять арифметические и условные выражения. Например, с точки зрения синтаксиса следующее выражение правильно:

```
k=1+a<b;
```

Если значение переменной a меньше Ь, то результат условного выражения будет равен 1, а значит, значение переменной k станет равным 2. В противном случае (если a больше или равно b) значение переменной k станет равным 1. Смешивая такие выражения, вы можете совершенно запутать вашу программу.

Цикл while

Формат инструкции while имеет следующий вид:

```
while(условное выражение) инструкция
```

Инструкция может быть составной. В результате сначала будет выполнено условное выражение, а потом, если результатом станет истина, — инструкция. Этот процесс будет повторяться до тех пор, пока результат выполнения условного выражения не станет ложным, тогда программа перейдет к выполнению следующей инструкции.

Если инструкция не является составной, ее не обязательно заключать в фигурные скобки и можно оставить на одной строке с инструкцией while.

Тем не менее я настоятельно рекомендую вам продолжать пользоваться фигурными скобками. Весьма распространена ошибка, когда, включая в цикл дополнительную инструкцию, вы забываете добавить фигурные скобки. Не забудьте также, что в этом случае в конце инструкции точка с запятой не ставится.

Цикл do...while

Формат инструкции do.. .while имеет следующий вид:

```
do инструкция whi le (условное выражение);
```

Сначала выполняется инструкция, а затем вычисляется условное выражение. Цикл завершается, когда условное выражение становится ложным.

Цикл for

Инструкция for обеспечивает полный контроль над итерациями и позволяет проводить инициализацию, проверку условия окончания цикла и подсчет. Основной формат инструкции for имеет вид:

```
for(начальное выражение; условие; конечное выражение) инструкция
```

Для этой инструкции истинно следующее:

- Начальное выражение выполняется только однажды, перед началом цикла.
- Условие представляет собой условное выражение, вычисляемое перед каждой итерацией. Если оно истинно, цикл продолжается.
- Конечное выражение выполняется в конце каждой итерации. Обычно оно служит для инкремента переменных и хранения числа итераций.
- Под инструкцией понимается любая инструкция (включая составную), которая выполняется при каждой итерации.

Выражения с умолчаниями

Выражения внутри скобок могут быть опущены. На примере цикла for нетрудно понять, что получается, если отсутствует начальное и (или) конечное выражение. Однако если опустить условие цикла, то это будет равносильно приданию ему значения истина! Приведенный ниже фрагмент программы представляет собой бесконечный цикл:

```
for(;;)
Sal.say("I am stuck!");
```

Запятые в выражениях

Б языке C++ любые выражения допускается отделять друг от друга запятыми. Например, в следующей инструкции перед выполнением цикла будут созданы и инициализированы целые переменные done и to_do соответственно значениями 0 и 10:

```
for( int done=0, to_do=10; done<=howmany; done++, to_do-)
```

В конце каждой итерации переменная done инкрементируется, а переменная to_do декрементируется.

Если нужно получить значение, получающееся в результате выполнения выражения, то берется то значение, которое относится к выражению, стоящему левее других. Например, в следующей инструкции переменной howmany будет присвоено значение 3:

```
howmany=3, 1, 2;
```



ВНИМАНИЕ Старайтесь не использовать разделенные запятыми составные выражения. Этим вы только запутаете свои программы, и в них будет трудно разобраться.

Объявление переменных

В инструкции for можно объявить и инициализировать переменную. Это полезно, когда переменная нужна только в процессе выполнения цикла. Например, рассмотрим следующую инструкцию:

```
for(int done=1; done<=howmany; done++)
```

Эта инструкция эквивалентна двум другим;

```
int done;
for(done=1; done<=howmany; done++)
```

Переменная, объявленная в начальном выражении инструкции for, считается объявленной до составной инструкции, определяющей цикл. Следовательно, ее область видимости простирается дальше последней инструкции цикла,



ПРИМЕЧАНИЕ В некоторых компиляторах переменная, объявленная в начальном выражении инструкции for, считается объявленной внутри цикла. В этих компиляторах область видимости такой переменной ограничена циклом.

Самостоятельная практика

О Какое значение отображает на экране приведенный ниже фрагмент программы? Почему?

```
tfinclude "franca.h"
void mainprog()
{.
   athlete Sal;
```

```
int done=5;
{
    int  done;
    for( int done=1; done<=10; done++)
    {
        Sal.ready();
        Sal.up();
    }
}
Sal.say(done);
}
```

О Определите значения переменных т и j.

- **for(int** m=0, intj=0; m<5;) ++m++;
- **for(int** m=0, intj=0; **m<5;** m++)j++;
- for(int j=0, int m=0; m++<5; m++;) j++;
- for(intj=0, **int** m=0; m<5; m=j++;) m++;

О Следующие упражнения предназначены для знатоков математики:

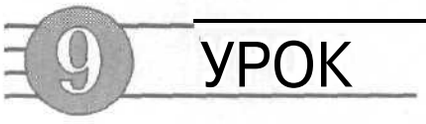
- Факториал целого неотрицательного числа N определяется произведением всех чисел от 1 до N. Факториал нуля равен 1. Составьте программу для считывания целого неотрицательного числа с клавиатуры, расчета и вывода на экран значения его факториала. Эта программа должна непрерывно запрашивать новые значения, пока ее работа не будет прервана. Для расчета факториалов вводимых значений используйте цикл.
- Для расчета факториалов вводимых значений используйте функцию.
- Хотя факториал и является целым числом, но значение типа int в качестве возвращаемого значения функции расчета факториала не слишком удобно. Почему для этой цели лучше использовать значение типа float или long?

Что нового мы узнали?

В этом уроке мы научились

0 Идентифицировать циклы с параметрами.

0 Создавать циклы в своих программах.



УРОК 9 Разработка базовых приложений

- Реализация торгового терминала
- Описание процесса разработки приложений

На этом уроке мы займемся вопросом разработки приложений и создадим простейший торговый терминал (похожий на тот, который вы можете увидеть в супермаркете). В дальнейшем, по мере обучения, вы будете постепенно усложнять свой терминал, а также создавать другие приложения.

Этот небольшой проект охватывает материал некоторых предыдущих уроков. Мы обсудим также ряд вопросов, возникающих при решении проблем реальной жизни, в которых значительная роль отводится разработчику.

Создание торгового терминала

Предположим, вам необходимо разработать программное обеспечение для реализации торгового терминала. С помощью этого программного обеспечения компьютер должен вести учет продаж, вычислять сдачу и обслуживать другие аспекты торговли.

Несмотря на то что ваши возможности как программиста пока ограничены, вы уже готовы решить эти проблемы.

В реальной жизни клиент (собственник товара) зачастую не совсем четко понимает, какого сорта программную продукцию он хочет от вас получить. Он знает только то, что вручную выполнять всю необходимую работу слишком хлопотно и что компьютер должен ему помочь. И именно разработчик (то есть вы) должен найти нужное решение.



ПРИМЕЧАНИЕ В большинстве торговых терминалов имеется специальное устройство для считывания штрих-кода товара, а также небольшой принтер. К сожалению, все что у нас есть, — это обычный компьютер и даже не обязательно с принтером! Пользователю вашего программного обеспечения (клерку) придется вводить атрибуты каждого товара с клавиатуры.

Программное обеспечение, которое вы собираетесь создавать, должно складывать стоимость всех товаров и (с учетом торговой надбавки) выводить ее на экран. Если бы покупатель делал единственную покупку, то решить проблему было бы просто. Для этого можно было воспользоваться одной из программ урока 7, а именно программой c3storel.cpp.

Однако теперь для обслуживания даже одного покупателя вам придется вводить и складывать цены разных товаров. Только тогда, когда будут введены все цены, вы сможете перейти к расчету тарифов, наценок, полной стоимости и т. д. Кроме того, программное обеспечение должно допускать обслуживание не одного, а нескольких покупателей.

Программное обеспечение для нескольких транзакций

Зная к настоящему времени базовые операции, можно попытаться составить общий алгоритм решения проблемы.

Поскольку процедура обслуживания покупателя выполняется многократно, решение проблемы можно описать следующим образом:

Пока имеются покупатели, повторять следующие инструкции:

Ввести и сложить цены товаров

Рассчитать и вывести на экран полную стоимость
с учетом торговой надбавки

Рассчитать и вывести сдачу

Это хотя и правильное, но все же не до конца точное описание того, что мы собираемся делать. Тем не менее для начала вполне подойдет.

Очевидно, что ввод и сложение цен подразумевают еще один цикл. Соответственно, вы можете либо добавить в алгоритм соответствующее описание, либо создать функцию для считывания и суммирования цен товаров, приобретаемых каждым покупателем. Таким образом, вы упростите основную программу и избавитесь от вложенного цикла.

Пусть возвращаемым значением такой функции (рассчитывающей полную стоимость приобретенных в данной покупке товаров без учета торговой надбавки) является число с плавающей точкой. Например:

```
float getitems();
```

Следующая задача — вывод этих знаний на экран.

Можно воспользоваться стандартным потоковым объектом Cout. Однако, как вы помните, все, что выводится на экран с помощью объекта Cout, будет стерто при следующем выводе. Для пользователя это может оказаться неудобным. С другой стороны, для вывода на экран конкретной информации: полной стоимости, полученной суммы, сдачи и т. д. можно создать несколько информационных рамок.

Воспользуемсярамками.

Информационные рамки

Рассмотрим возможные варианты информационных рамок:

- Current total (текущая сумма) — отображает накапливаемую (текущую) стоимость товаров без учета надбавки.
- Sales total (общая сумма) — отображает полную сумму, которую должен заплатить покупатель (включая надбавку).

- Amount tendered (уплаченная сумма) — отображает полную сумму, которую уплатил покупатель,
- Change (сдача) — отображает сдачу, которую следует вернуть покупателю.

Рамки можно объявить следующим образом:

```
Box Cur_total("Current total:");
Box total("Total Sale:");
Box Amount("Tendered:");
Box Change("Change due:");
```

Эти объявления нужно разместить в начале программы.

Теперь частично в виде алгоритма, а частично в виде программы можно написать полный цикл обслуживания каждого покупателя:

```
// Ввод и накапливание стоимости каждой покупки
// Учет торговой надбавки:
saletotal=getitems()*salestax;
// Вывод полной стоимости товара:
total.say(saletotal);
// Запрос на ввод уплаченной суммы и вывод ее на экран:
amount=ask("Enter amount tenderd:");
Amount.say(amount);
// Расчет сдачи и вывод ее на экран:
change=amount-saletotal;
Change.say(change);
```

Приведенный выше фрагмент программы предназначен для обслуживания каждого отдельного покупателя. Далее его можно включить в цикл, но как при этом определить число итераций?

Вспомните, что завершить цикл можно несколькими способами. Я надеюсь, вы понимаете, что определить заранее число покупателей невозможно. Наиболее подходящий вариант в нашем случае — после каждой итерации выдавать запрос о необходимости обслуживания следующего покупателя. Это очень просто:

```
while(yesno ("Is there another customer?"))
```

Если при этом вы хотите учитывать число покупателей, можно воспользоваться циклом for и инкрементировать переменную customer:

```
for(;yesno ("is there another customer?"); customer++)
```

Начальные условия

Для полного задания цикла необходимо определить начальные условия. Цикл обслуживания каждого покупателя сам по себе начальных условий не требует. Одна-

ко инициализации нужна функции getitems() для вычисления стоимости товара в каждой покупке. Не забывайте, что цикл должен выполняться для каждого покупателя. При этом все, что ранее было вычислено и/или выведено на экран, но относилось к предыдущему покупателю, должно быть стерто. Эта означает, что необходимо инициализировать все информационные рамки.

```
saletotal=0;
amount=0;
change=0;
Cur_total=0;
Saletotal.say(saletotal);
Amount.say(amount);
Change.say(change);
Cur_total.say(0);
```

Эта последовательность инструкций не только инициализирует все значения, но и выводит каждое из них в соответствующей рамке. Вместо того чтобы писать в рамках нули, их содержимое можно просто стирать. Например:

```
Amount.say("");
```

Функция getitems()

Функция getitems() должна выводить на экран запрос о стоимости очередного товара, приобретенного конкретным покупателем, накапливать вводимые значения и в конце концов вывести накопленный результат на экран. Такой подход можно использовать для организации цикла:

```
float getitems()
{
   float total=0;
   for(int item=1; yesno("another item?"); item++)
   {
      total= total+ask("Enter item price:");
   }
   return total;
}
```

Вероятно, было бы неплохо, как в настоящем кассовом аппарате, выводить на экран накапливаемую (текущую) стоимость товаров.

Однако такая реализация сопряжена с рядом проблем. Если для отображения текущей стоимости товаров объявить новую рамку, она будет создаваться при каждом вызове функции. Хотя после завершения выполнения функции такая рамка исчезнет, каждая следующая будет появляться в другом месте экрана.

Информационная рамка с параметрами

Чтобы решить эту проблему, нужно объявить информационную рамку вне области видимости функции. Можно объявить рамку в основной функции mainprog() и передать ее функции getitems() в качестве аргумента, причем лучше это сделать по ссылке. Вспомните, если аргумент (в данном случае рамка) передается по ссылке, то функция не создает его копию (в данном случае новую рамку на новом месте). Вместо копии будет использоваться объявленная в программе исходная рамка. Можете самостоятельно попробовать передать рамку по значению и посмотреть, что получится.

Обновленная версия функции имеет следующий вид:

```
float getitems(Box &display)
{
   floattotal=0;
   for(int item=1; yesno("anothe item?"); item++)
   {
      total=total+ask("Enter item price:");
      display.say(total);
   }
   return total;
}
```

Соответствующую инструкцию в функции mainprogQ также необходимо изменить:

```
saletotal=getitems(Cur_total)*(1+tax);
```

Теперь вернемся к торговой надбавке. Очевидно, ее можно определить как константу, но тогда при каждом изменении надбавки пришлось бы переделывать программу. Это, согласитесь, не очень удобно. Для решения проблемы в начале программы можно запрашивать значение наценки и сохранять его в течение всего времени работы программы (то есть в течение рабочего дня). Это легко можно сделать с помощью следующей инструкции:

```
tax=ask("Enter the sales tax %")/100;
```

Полный листинг программы

Ниже представлен полный листинг модифицированной программы c3sale2.cpp:

```
#include "franca.h"      //c3sale2.cpp
// Программа создания торгового терминала:
//    Получает информацию о торговой надбавке
//    Считывает стоимость каждого товара
//    Находит полную стоимость каждой покупки
//    Вычисляет сдачу
```

продолжение

```
float getitems(Box &display)
{
   float total=0;
   for(int item=1; yesno("another item?"); item++)
   {
      total=total+ask("Enter item price:");
      display.say(total);
   }
   return total;
}
void mainprog()
{
   Box Cur_total("Current total:");
   Box total("Total Sale:");
   Box Amount("Tendered:");
   Box Change("Change due:");
   float tax, amount, saletotal, change;
   dailytotal=0;
   tax=ask("Enter the sales tax %")/100.;
   for( int customer=1; yesno("Is there another customer?"); customer++)
   {
      // Начало обслуживания одного покупателя
      saletotal=0;
      amount=0;
      change=0;
      Saletotal.say(saletotal);
      Amount.say(amount);
      Change.say(change);
      saletotal=getitems(Cur_total)*(1+tax);
      Saletotal.say(saletotal);
      amount=ask("Enter amount tendered:");
      Amount.say(amount);
      change=-amount-saletotal;
      Change.say(change);
   }
}
```

На рис. 9.1 показан результат обслуживания покупателя, купившего товаров на сумму $ 12.24 (без учета торговой надбавки в 8.25%).

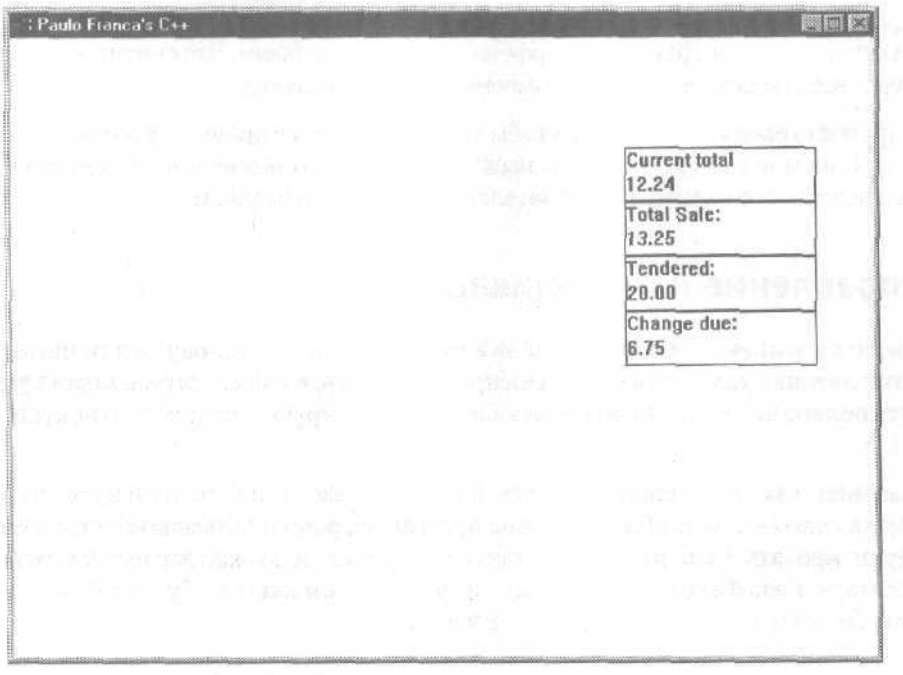


Рис. 9.1. *Ваш вариант торгового терминала*

Описание процесса разработки приложений

Обычно процесс разработки приложений далеко не такой гладкий, как может показаться на первый взгляд. Я мог бы представить этапы этого процесса в более удобной для вас последовательности, но знаю, что при создании собственного проекта это вам не понадобится. Например, когда вы собираетесь создать какую-нибудь функцию (скажем, getitems()), вы можете не знать всего, что для нее потребуется. В нашем случае я попытался показать, что необходимость в дополнительном параметре возникла только после того, как код функции был написан. Это характерная ситуация для реальной жизни. Ничего страшного. Просто вернитесь назад и внесите необходимые поправки.

Чем лучше вы понимаете проблему и имеющиеся в вашем распоряжении инструменты, тем лучше вы справитесь с решением.

Пользовательский интерфейс

Не следует пренебрегать пользовательским интерфейсом. *Пользовательский интерфейс* (user interface) — это средства общения пользователя с вашей программой, ко-

торые могут включать в себя изображения, звуки и текст. Ориентируйтесь на среднего пользователя. Делайте интерфейс простым и удобным. Это снизит вероятность пользовательских ошибок и, несомненно, ему (пользователю) понравится.

С другой стороны, старайтесь, чтобы ваша программа в процессе работы не задавала слишком много вопросов и не выводила на экран объяснения и без того очевидных вещей. Это вашему пользователю может быстро надоесть.

Исправления и модификации

После выполнения основной задачи вам могут прийти в голову те или иные идеи, позволяющие улучшить ваше решение. Например, в нашем случае можно учитывать полное число купленных товаров, дневную выручку, количество покупателей и т. д.

Наконец, сам пользователь может попросить вас в чем-то изменить решение. Сколь совершенна ни была бы ваша программа, рано или поздно ее придется модернизировать. Если предвидеть такие изменения, то можно заранее к ним подготовиться. Разрабатывая программу, помните о возможности будущей модернизации. Делайте ее понятной и легко изменяемой.

Самостоятельная практика

О Модифицируйте программу торгового терминала, чтобы:

- Выводить на экран число товаров, приобретенных каждым покупателем.
- После обслуживания последнего покупателя вывести на экран количество покупателей и дневную выручку.

Что нового мы узнали?

В этом уроке мы научились

0 Разрабатывать простые приложения.

0 Разрабатывать подходящий пользовательский интерфейс приложений.

И Понимать некоторые аспекты создания приложений.

Часть IV

Условия

Часто выбор того или иного варианта действий зависит от некоторых условий. В части IV вы узнаете, как встраивать такие условия в ваши программы. Вы также научитесь пользоваться *рекурсией*, специальным приемом, при использовании которого решение проблемы идет как бы в обратном направлении. Кроме того, чтобы попрактиковаться в создании приложений, вы поработаете над небольшим проектом.



УРОК 10 Условные инструкции

- **Условная инструкция if**
- Условная инструкция **if ... else**
- Задание и проверка условий
- Выход из циклов
- Второй этап создания циклов
- Выбор варианта

На уроке 10 вы научитесь встраивать в свои программы возможность выбора. В зависимости от выполнения условия выбирается тот или иной вариант продолжения программы. Вы узнаете, как задать и использовать условия в инструкции if, чтобы выполнить заданный фрагмент программного кода.

Кроме того, вы познакомитесь с условиями прерывания циклов и попрактикуетесь в их создании. В этом вам будут помогать объекты типа Robot.

Инструкция if

Как правило, выбор подразумевает условие, за которым следует действие. Например, если идет дождь, нужно взять зонтик. В этом случае условие — *идет дождь,* а действие — *взять зонтик*. Действие совершается только при выполнении условия. Условие лежит в основе выбора. В предыдущем примере, чтобы сделать выбор, достаточно знать только, идет ли дождь.

При создании программ используется аналогичный процесс принятия решений. Большинство решений основаны на условии, в зависимости от которого и выполняются действия. Инструкция if — это основной механизм принятия подобных решений в программах.

Начнем с простейшего случая: делать или не делать. Как, например, в хорошо вам знакомой ситуации:

```
Если у вас есть деньги:
    Идти в кино
```

Здесь действие *идти в кино* выполняется только в том случае, если истинно условие *(есть деньги)*. В противном случае ничего не произойдет, кроме того, конечно, что вы можете перейти к выполнению следующей инструкции, например *идти спать*.

Более полное описание проблемы имеет следующий вид (рис. 10.1, а):

```
Если у вас есть деньги:
    Идти в кино
Идти спать
```

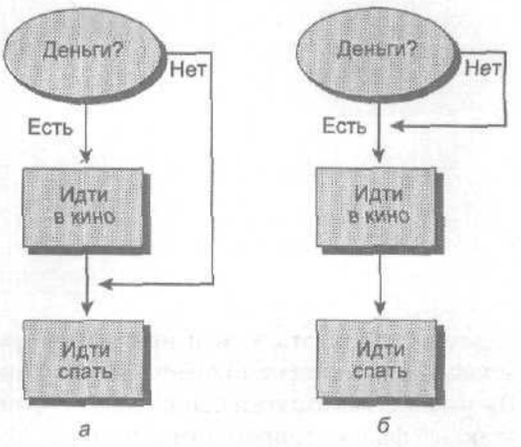


Рис.10.1. *Поход в кино*

Важно понять, что в данном случае спать вы все равно пойдете. А вот действие *идти в кино* можно либо выполнить, либо нет. Как мы это узнали? Благодаря отступам. Рассмотрим следующие три команды:

Если у вас есть деньги:

Идти в кино

Идти спать

Из-за отсутствия отступа можно предположить, что мы сначала проверяем наши карманы, а потом, вне зависимости от результата, сначала идем в кино, а затем — спать (рис. 10.1, *б).* А все потому, что по нашему соглашению составная инструкция, на выполнение которой влияет условие, должна сдвигаться. В программах на C++ помимо отступов нужно использовать фигурные скобки.

Рассмотренный нами тип принятия решения состоит в выборе между делать и не делать. Это напоминает знакомую нам ситуацию с роботом, мы предлагали ему делать или не делать шаг вперед в зависимости от того, имеется или нет перед ним стена. Эта процедура описывалась следующим циклом:

```
while(Tracer.seenowall())
{
   Tracer, step();
}
```

С помощью инструкции while робот решает, продолжать или нет выполнение цикла в зависимости от наличия стены перед ним. А если вам необходимо, чтобы робот делал всего один шаг, а не ходил до тех пор, пока не упрется в стену? Видимо, вы дадите ему примерно такие инструкции:

Если впереди нет стены;

Сделать шаг вперед

Робот проверит условие *впереди нет стены* и, если оно окажется верным, выполнит действие *сделать шаг вперед*. А что, если условие не верно (если впереди стена)? Просто ничего не произойдет. Робот не сделает шаг вперед, и программа перейдет к следующей инструкции.

В C++ указанный алгоритм можно выполнить с помощью инструкции if (рис. 10.2).

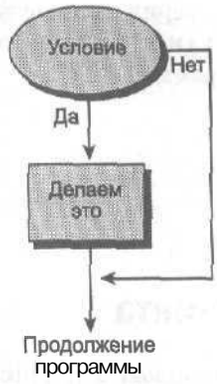


Рис. 10.2, *Действие инструкции if*

Синтаксис инструкции if выглядит следующим образом:

```
if(условие)
{
   инструкции     // Делаем это
}
```

В нашем примере это будет выглядеть так:

```
if(Tracer.seenowall())
{
   Tracer.step();
}
```

Обратите внимание на формат инструкции if:

- Указывается ключевое слово if.
- Составляется условие, представляющее собой условное выражение, заключенное в круглые скобки.
- Последовательность выполняемых инструкций сдвигается вправо и заключается в фигурные скобки.



ВНИМАНИЕ Ни после условия, ни после составной инструкции точка с запятой не ставится. Точка с запятой после условия показывает компилятору, что *ничего* делать не надо. Точка с запятой после составной инструкции — это безвредная ошибка.



СОВЕТ Если последовательность, которую необходимо выполнить, состоит всего из одной инструкции, ее не обязательно заключать в фигурные скобки. Однако я настойчиво советую привыкнуть постоянно ставить скобки.

Если вы хотите, чтобы робот проверил наличие впереди стены и, если есть, пометил бы соответствующую ячейку (и больше ничего не делал), то можете использовать следующий фрагмент программы:

```
if(Tracer.seewall())
{
   Tracer.mark();
}
```

Первый пример лабиринта

Создадим для нашего робота несколько функций. Одна из них, функция до(), будет заставлять робота делать шаг вперед после проверки, является ли находящаяся впереди ячейка пустой. Если ячейка занята, никакого действия не выполняется. Вторая функция crossing(), изучив ситуацию непосредственно вокруг робота, вычислит, сколько таких ячеек свободно. Другими словами, функция crossing() будет вычислять количество возможных направлений (N, S, E, W) движения.

Алгоритм функции go() предельно прост:

Если впереди нет стены:

Сделать шаг вперед

Обратите внимание, если впереди есть стена, никаких действий выполнено не будет. Возможная реализация функции go():

```
void go()
{
   if(Tracer.seenowall())
   {
     \\ Следующая инструкция выполняется только в тон случае,
     \\ если свободна находящаяся перед роботом ячейка:
     Tracer.step();
   }
}
```

Чтобы разнообразить функцию, давайте добавим в нее следующие возможности:

- Закрашивать ячейки разными цветами.
- Заставлять робота произносить «ОК», когда путь впереди открыт.

Если использовать светло-голубой цвет (код 4), то модифицированная версия функции примет следующий вид:

```
void go()
{
   if(Tracer.seenowall())
   {
     \\ Следующие инструкции выполняются только в том случае,
     \\ если свободна находящаяся перед роботом ячейка:
      Tracer.mark(4);
      Tracer.step();
      Tracer. say("OK");
   }
}
```

Вторая функция crossing() также достаточно проста. Ее алгоритм складывается из использования переменной для вычисления подходящих направлений (howmany) и просмотра очередной ячейки, чтобы определить, не занята ли она. Если она свободна, то к переменной howmany прибавляется единица. Если выполнить эту процедуру для всех четырех сторон света, в переменной howmany окажется сумма всех свободных направлений.

Целые числа и лабиринт

Если при объявлении объекта типа Robot вслед за именем объекта указать в скобках любое целое число (например, Robot Tracer(1);), робот окажется в лабиринте. Как его положение и направление, так и расположение стен вначале неизвестны. Вы можете позабавиться, исследуя лабиринт.

Если при объявлении объекта типа Robot вслед за именем объекта ничего не указано (например, Robot Tracer;), робот окажется в пустой комнате, повернутым на восток.

Объявляйте в своих программах только один объект типа Robot и, кроме того, объявляйте его глобально, а не внутри функций. В противном случае вам пришлось бы передавать его функциям в качестве аргумента.

Самостоятельная практика

О Напишите функцию int crossing() для определения суммы всех возможных направлений движения робота. Проверьте эту функцию с помощью следующей программы:

```
include "franca.h"
Robot Tracer(1);
void mainprog()
{
   Tracer.say(crossing());
}
```

Первый пример магазина

Давайте вернемся к проблеме вычисления сдачи за покупку товаров. Основная функция показана ниже.

```
include "franca.h"
void mainprog()       //c4change.cpp
{
   float price, amount, change;
   float tax=0.08;
   amount=ask("Enter the amount:");
   price=ask("Enter the price")*(1+tax);
   change=amount-price;
   Count<<change;
}
```

Эта программа считывает предлагаемую сумму и стоимость товара, а после начисления наценки определяет сдачу, значение которой выводится на экран. Заметим, что достаточность предлагаемой суммы этой программой не контролируется.

Можно ввести в программу соответствующую проверку и, когда предлагаемая сумма мала, предупреждать об этом продавца. Например:

```
include"franca.h"
void mainprog()
{
   float price, amount, change;
   float tax=0.08;
   amount=ask("Enter the amount:");
   price=ask("Enter the price")*(1+tax);
   if(amount<price)
   {
      Clock time;
      Count<<"This is not enough!";
      time.wait(5);
   }
   change=amount-price;
   Count<<change;
}
```

Единственным изменением в работе программы является сообщение о том, что предлагаемая сумма недостаточна (инструкция `Count<<«This is not enough!»;`). Пятисекундная пауза (инструкция `time.wait(5);`) необходима, чтобы успеть прочитать сообщение.

Инструкция if...else

Другой тип принятия решения связан с выбором между двумя альтернативными последовательностями действий:

Взять кредитную карточку

Если рынок открыт:

Продать дюжину яиц

Получить двадцать долларов

В противном случае:

Снять двадцать долларов с кредитной карточки

Заполнить газовый баллон

В зависимости от того, открыт рынок или нет, вы выбираете одну из двух альтернатив:

- Продать дюжину яиц и получить двадцать долларов (на рынке).
- Идти в банкомет и снять двадцать долларов с кредитной карточки.

Вы не будете делать обе эти вещи вместе!

Вне зависимости от того, какой вариант вы выбрали, газовый баллон будет наполнен, поскольку условие не влияет на это действие. Самое важное здесь то, что мы определяем два альтернативных варианта развития событий (рис. 10.3).

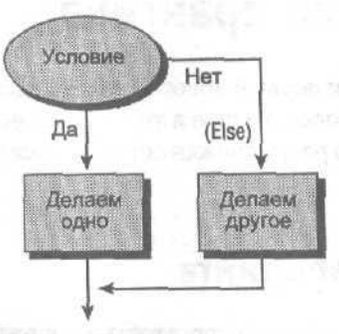


Рис. 10.3. *Выбор двух альтернативных вариантов развития событий*

Соответствующий синтаксис аналогичен синтаксису инструкции if, за исключением ключевого слова else, которое определяет действия, когда условие не выполняется.

```
if (условие)
{
   инструкции // Делаем одно
}
```

```
else
{
   инструкции // Делаем другое
}
```

Предположим, вы хотите, чтобы робот закрашивал ячейки в черный или зеленый цвет в зависимости от того, есть перед ним стена или нет:

```
int black=7, green=2;
if (Tracer.seewall())      // Есть впереди стена? Если да, то:
{
   Tracer.mark(black);     // Закрашиваем черным цветом
}
else
{
   Tracer.mark(green);     // Иначе: закрашиваем зеленым цветом
}
```

Поскольку каждый вариант реализуется всего одной инструкцией, можно обойтись без фигурных скобок. Ту же конструкцию можно написать следующим образом:

```
if(Tracer.seewall()) Tracer.mark(black);
else Tracer.mark(green);
```

Самостоятельная практика

- Заставьте робота найти первый поворот налево. Помните, он может определить, что слева имеется поворот, только в том случае, если повернется влево! После обнаружения поворота робот должен остановиться и сообщить количество пройденных шагов.

Второй пример лабиринта

Переработаем программу c4search.cpp, чтобы улучшить ее визуальный интерфейс. Будем помечать ячейки, в которых была обнаружена стена, другим цветом. Нам понадобится только изменить функцию до(). Если вы вспомните конструкцию if...else, то обнаружите, что в функции до() часть *делаем одно* уже присутствует. Нам необходимо включить туда часть *делаем другое:*

```
if (условие)
{
   // Делаем одно
}
```

```
else
{
   // Делаем другое
}
```

Сможете сделать это сами? Я думаю, да!

Самостоятельная практика

О Дополните функцию go() программы c4search.cpp так, чтобы она выполняла следующие действия, когда очередная ячейка является стеной:

- Закрашивала следующую ячейку черным цветом.
- Выдавала сообщение «OUCH!».

Третий пример лабиринта

Другой интересный пример состоит в том, чтобы заставить робота идти по коридору и закрашивать как свободные, так и занятые ячейки. Для этого в каждой ячейке робот должен повторить следующую последовательность действий:

Повернуть налево и проверить, есть ли там стена

Если да:

Закрасить ячейку черным

Иначе:

Закрасить ячейку зеленым

Повернуться в исходном направлении

Ту же проблему можно описать следующим образом:

Повторять до тех пор, пока впереди стена:

Проверить справа

Проверить слева

Сделать шаг вперед

Очевидно, необходимо пояснить, что мы понимаем под словами *проверить справа* и *проверить слева*! Вот, например, что означает *проверить справа:*

Повернуть направо

Если впереди стена:

Закрасить ячейку черным

Иначе:

Закрасить ячейку зеленым

Повернуть налево

Я надеюсь, теперь вы сами поймете, что означает *проверить слева*?

Можете определить роль конструкции if...else в этом алгоритме? В зависимости от того, занята ячейка или свободна, робот закрашивает ее в черный или зеленый цвет. Он выбирает либо то, либо другое действие. Если условие истинно, то выполняется действие, которое определяет ключевое слово if, а действие, которое определяет ключевое слово else, опускается. И наоборот, если условие ложно, опускается действие, которое определяет ключевое слово if, и выполняется действие, которое определяет ключевое слово else.

Фрагмент программы, описывающий проверку справа, может выглядеть так:

```
Tracer.right();
if(Tracer.seenowall())
{
  Tracer.mark(green);
}
else
{
  Tracer.mark(black);
}
Tracer.left();
```

Может быть, имеет смысл создать для такой проверки функцию check:

```
void check()
{
  if(Tracer.seenowall()) Tracer.mark(green);
  else Tracer.mark(black);
}
```

Ваша программа могла бы выглядеть следующим образом:

```
#include "franca.h"
Robot Tracer(1);
void mainprog()
{
  while(Tracer.seenowall) // Повторять, пока впереди стена
  {
    Tracer.left();        // Проверка слева
    check();
```

```
        Tracer.right();
        Tracer.right();        // Проверка справа
        checkC);
        Tracer.left();
        Tracer.step();         // Шаг вперед
    }
}
```

Второй пример магазина

Модернизируем программу вычисления сдачи c4change.cpp. Эту программу мы уже модернизировали, чтобы можно было проверить, достаточна ли предложенная сумма для оплаты товара,

Лучшее решение подразумевает использование конструкции if...else. Другими словами, если предлагаемая сумма недостаточна, то сделка отменяется.

Одно из возможных решений приведено ниже:

```
include "franca.h"
void mainprog()        //c4change.cpp
{
   float price, amount, change;
   float tax=0.08;
   amount=ask("Enter the amount:");
   price=ask("Enter the price")*(1+tax);
   if(amount<price)
   {
      Count<<"This is not enough!";
   }
   else
   {
   change=amount-price;
   Count<<change;
   }
}
```

В этом случае на экран выводится либо значение сдачи, либо сообщение, что предлагаемой суммы недостаточно, но ни то и другое вместе.

Задание условий

Условия используются при принятии решений. Мы уже знакомы с простыми условиями, основанными на сравнении двух чисел. Например, следующее условие проверяет, является ли значение переменной count меньшим или равным 15:

```
count<=15
```

Условие базируется на сравнении значений и описывается посредством условного выражения.

Условные выражения

Простейшее условное выражение состоит из двух частей, разделенных оператором отношения. Этими частями могут быть:

- Переменные. Например, следующее выражение проверяет, будет ли значение переменной count равно значению переменной howmany:

```
count == howmany
```

- Константы (условное выражение, обе части которого представляют собой константы, хотя и допустимо, но бессмысленно). Например, следующее выражение проверяет, будет ли значение переменной howmany *не* равно нулю:

```
howmany ! = 0
```

- Результаты, возвращаемые функцией. Например, следующее выражение проверяет, будет ли время, возвращаемое объектом myclock в ответ на сообщение time(), меньше или равным 20:

```
myclock.time() <= 20
```

- Арифметические выражения. Например:

```
howmany - steps > 10
```



ПРИМЕЧАНИЕ Объекты также можно использовать в условных выражениях, но только в том случае, если оператор отношения определен для объектов этого класса.

В отличие от оператора присваивания, в операторе отношения левую и правую части можно менять местами. Например, условие count<=15 эквивалентно условию:

```
15>count
```

Операторы отношения

Операторами отношения, о которых мы уже говорили на уроке 5, являются:

== равно

!= неравно

< меньше

<= меньше или равно

> больше

>= больше или равно

Результаты

Компьютер вычисляет условное выражение и выдает результат. Для практических целей результат можно рассматривать как истину либо ложь. Например, если текущее значение переменной count равно 15, то следующее выражение истинно:

```
count<=15
```



ВНИМАНИЕ Напомню вам еще раз самое опасное выражение: if(a=b). Это не условие, а присваивание! В условном выражении используются два знака равенства (==).

Составные выражения

Нам часто приходится выносить свое решение на базе нескольких выражений. Например, гимнаст может закончить занятия либо по истечении 5 минут, либо после 50 упражнений. В этом случае мы получаем два условия:

```
count>=50
time>=300
```

Когда количество условий больше (или равно) двух, мы должны описать порядок их выполнения. В данном случае мы хотим, чтобы выполнялось либо одно, либо другое условие. В других случаях может потребоваться, чтобы были выполнены оба условия (одно и другое).

Логические операторы

Логические операторы используются для объединения двух или более условий. Имеются следующие логические операторы:

- && — оператор конъюнкции (оба условия должны быть истинны).
- || — оператор дизъюнкции (либо одно, либо другое, либо оба условия вместе должны быть истинны).
- ! — оператор отрицания (меняет результат выполнения условного выражения на обратный).

Оператор отрицания является унарным. Он может ставиться перед выражением, меняя его значение на обратное. Остальные операторы являются бинарными, они

располагаются между двумя выражениями. Выражение должно заключаться в круглые скобки.

Примеры:

```
(count>=15)||(myclock.time()>300)
```

Либо значение переменной count больше или равно 15, либо возвращаемое значение функции time() больше 300.

```
(count>=15)&&(!(myclock.time()<120)
```

Значение переменной count больше или равно 15, и возвращаемое значение функции time() больше 120.

```
(count>15)&&(!(myclock.time()<240))
```

Значение переменной count больше 15, и возвращаемое значение функции time() *не* меньше 240.

Приоритет

Если выполняется более двух выражений, важно знать приоритет операторов:

- Высший приоритет имеет оператор отрицания !.
- Следующий приоритет имеет оператор конъюнкции &&.
- Наименьший приоритет имеет оператор дизъюнкции ||.

Для изменения приоритета используются скобки. Например:

```
(myclock.time()>300)||(myclock.time()>120)&&(count>=50)
```

Здесь первым выполняется оператор &&, а за ним ||. Если это выражение определяет, в какой ситуации гимнаст может закончить выполнение упражнений, то это случится, если:

время больше 300

или

время больше 120 и количество упражнений больше или равно 50

Вложенные условия

В некоторых ситуациях, чтобы решить, какое действие должно быть выполнено, необходимо проверить более одного условия. Это происходит, когда внутрь одной инструкции if вкладывается другая инструкция if. Рассмотрим следующий пример:

```
// Проверка справа:
Tracer.right();
if(Tracer.seewall())
{
   // Проверка слева:
   Tracer.left();
   Tracer.left();
   if(Tracer.seewall())
   {
      Tracer.say("corridor");
   }
}
```

Б рассмотренном примере имеются два уровня вложения. Если выражение в первой инструкции if ложно (стены нет), то вторая инструкция if не выполняется. Этот фрагмент программы определяет, есть ли стены по обе стороны от объекта Tracer. Обратите внимание, что робот сначала поворачивается направо и проверяет правую сторону. Если стены там нет, то нет необходимости проверять левую сторону, и тест заканчивается. Если же справа есть стена, то робот должен повернуться к левой ячейке (для этого нужно сделать два поворота) и проверить ее. Если и здесь имеется стена, робот сообщит, что находится в коридоре, где справа и слева стены.

Представьте себя на месте робота и попытайтесь понять работу этого алгоритма.

Чем больше в вашей программе вложенных условий, тем она сложнее и интереснее. Правильный и четкий отступ, равно как и подходящий комментарий, сделают вашу программу более удобочитаемой. Если программа по-прежнему выглядит сложной, имеет смысл обратиться к функциям.

Одним из недостатков синтаксиса языка C++ является то, что у компилятора нет возможности определить, имеется ли за инструкцией if соответствующая ей часть else.

Рассмотрим, например, следующие фрагменты программ:

```
k=0;
   if(a<0)
      if(b<0) k=1;
   else k=2;
```

и

```
k=0;
   if(a<0)
      if(b<0) k=1;
      else k=2;
```

Поскольку отступы для компилятора роли не играют, оба фрагмента для него идентичны. Однако нам отступ указывает, что фрагменты отличаются. В первом примере, если значение a больше или равно нулю, выполняется инструкция else и переменной k присваивается значение 2. Во втором примере инструкция else является частью внутренней инструкции if, и, таким образом, если значение a больше или равно нулю, то внутренняя инструкция if пропускается и переменная k остается равной нулю.

Очевидно, правильным может быть только один вариант, поскольку, как уже говорилось, с точки зрения компилятора оба фрагмента идентичны. Что выполнит компилятор?

Правилен второй фрагмент, поскольку в этом фрагменте ключевое слово else ближе к внутренней инструкции if.



ПРИМЕЧАНИЕ Ключевое слово else всегда является частью ближайшей инструкции if.

Если вы хотите, чтобы ключевое слово else относилось к первой инструкции if, нужно добавить в программу строку с пустой инструкцией else:

```
k=0;
   if(a<0)
      if(b<0) k=1;
      else;
   else k=2;
```

Другой, более элегантный прием позволяет обойтись без части else во второй инструкции if:

```
k=0;
   if(a<0)
   {
      if(b<0) k=1;
   }
   else k=2;
```

Выход из циклов

Иногда, выполняя цикл, вы по какой-то причине решаете прервать итерации.

Представьте, что нашему старому другу Сэлу нужно выполнить 50 подскоков. Он прыгает — 1, 2, 3, 4, — и вдруг звучит пожарная тревога. Как поступить: закончить упражнение или убежать? Конечно, убежать! Так и мы иногда вынуждены прерывать выполнение цикла.

Инструкция break

В C++ для прерывания цикла используется инструкция break. Она позволяет выйти из текущего цикла так, как будто на инструкции break заканчиваются все итерации. Рисунок 10.4 иллюстрирует эту ситуацию.

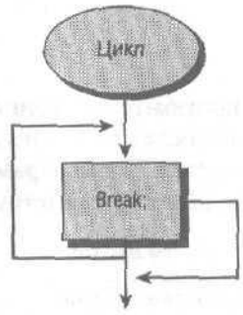


Рис. 10.4: *Инструкция break*

Допустим, вы создали цикл с условием выполнения определенного числа итераций. Однако по ходу выполнения цикла вы проверяете еще одно условие: *Здание горит?*, при выполнении которого необходимо выйти из цикла. При этом вы не только прекращаете итерации, но вдобавок пропускаете некоторую часть цикла.

Проиллюстрируем это следующим образом. Пусть Сэл выполняет упражнения и, кроме того, проверяет, сколько на это требуется времени:

Повторить следующие инструкции 50 раз:

Если прошло более 50 секунд:

Остановиться

Выполнить упражнение

Ниже представлен соответствующий фрагмент программного кода:

```
athlete Sal;
Clock watch;
for(int times=1; times<=50; times++)
{
   if(watch.time()>50.) break;
   JumpJack(Sal);
}
```

Не забыли ли мы ключевое слово else? Ведь у нас два варианта действий: либо прервать упражнение, либо его продолжить, что, как мы знаем, описывается конструкцией if. . .else, а не инструкцией if. Другими словами, вы наверное предложили бы следующую программу:

```
athlete Sal;
Clock watch;
for(int times=1; times<=50; times++)
{
   if(watch.time()>50.) break;
   else JumpJack(Sal);
}
```

Это решение хотя правильно, но избыточно. При выполнении инструкции break все инструкции, расположенные после нее вплоть до конца цикла (до закрывающей фигурной скобки), пропускаются- Таким образом, функция JmpJack() вызывается только в том случае, если не выполнена инструкция break.

В следующем примере робот делает 10 шагов:

Повторить следующую инструкцию 10 раз;

Сделать шаг

Чтобы робот не ударился о стену, пусть он останавливается, когда впереди стена:

Повторить следующую инструкцию 10 раз:

Если впереди стена:

Прекратить повторения

Сделать шаг

Готово!

Что мы хотим от робота? Мы хотим, чтобы перед стеной он игнорировал условия выполнения цикла и сразу перешел к следующей инструкции, а именно к инструкции *Готово!*. Для этого опять воспользуемся инструкцией break:

```
int stepcount;
for(stepcount=1, stepcount<=10, stepcount++)
{
   if(Tracer.seewall())
   {
      break;
   }
   Tracer.step();
}
```

Инструкция continue

В некоторых случаях может понадобиться прервать выполнение текущей итерации и сразу перейти к следующей. Для этого используется инструкция continue.

В отличие от инструкции break, инструкция continue прерывает выполнение только текущей итерации, а не всего цикла.



ПРИМЕЧАНИЕ Если вы знакомы с настольными играми, в которых фишки движутся к цели по клеточкам, то можете сравнить инструкцию break с клеточкой, попав на которую фишка выбывает из игры, а инструкцию continue с клеточкой, попав на которую фишка возвращается на начальную позицию.

В качестве примера допустим, что один из наших гимнастов 50 раз выполняет упражнения:

Подскок

Вправо-влево

Предположим, что следующие функции определены:

```
JumpJack(athlete)
leftright(athlete)
```

Тогда программа будет выглядеть так:

```
athlete Kim;
Clock watch;
for(int times=1; times<=50; times++)
{
   JumpJack(Kim);
   leftright(Kim);
}
Kim.say("Done!");
```

Предположим теперь, что если упражнения длятся больше 15 секунд, то гимнаст Ким будет пропускать второе упражнение (вправо-влево). Для контроля времени используем инструкцию if. Ниже представлен новый вариант программы:

```
athlete Kim;
Clock watch;
for(int times=1; times<=50; times++)
{
   JumpJack(Kim);
   if(wath.time()>15.) continue;
   leftright(Kim);
}
Kim.say("Done!");
```

При каждой итерации Ким выполняет подскок и затем смотрит на часы. Если 15 секунд еще не прошло, он продолжает этап до конца и выполняет упражнение вправо-влево. Однако если 15 секунд истекли, выполняется инструкция continue. В результате текущая итерация прерывается, но тем не менее значение переменной times увеличивается на 1 и начинается следующая итерация. Поскольку теперь время, прошедшее с момента начала упражнений, будет всегда больше 15 секунд, Ким будет выполнять только подскок и пропускать упражнение вправо-влево. Если бы вместо инструкции continue использовалась инструкция break, то прекратились бы все итерации и Ким закончил бы упражнения.

Второй этап создания циклов

В разделе «Первый этап создания циклов» урока 8 мы не пользовались инструкциями continue и break. С их помощью можно либо пропускать оставшуюся часть цикла и начинать итерации заново, либо вообще выходить из цикла.

В качестве примера вернемся к программе c3avgrd.cpp, в которой мы вычисляли среднюю успеваемость в классе. Основной цикл имеет следующий вид:

```
for(int which=0; yesno("Input another grade?"); which++)
{
   grade=ask("Enter next grade:");
   total=total+grade;
}
```

Допустим, что вашему преподавателю надоело каждый раз реагировать на предлагаемый функцией yesno() запрос о необходимости ввода следующего значения. В конце концов, если в классе 40 студентов, то любому может надоесть 40 раз нажимать на кнопку мыши!

Альтернативой является использование сигнального значения. Например, поскольку значение любого балла положительно, то с помощью отрицательного значения можно сигнализировать компьютеру об окончании ввода.

Новый цикл примет следующий вид:

```
for(int which=1; ; which++)
{
   grade=ask("Enter next grade:");
   if(grade<0) break;
   total=total+grade;
}
```

Ввод отрицательного значения балла приведет к немедленному прекращению цикла. Это отрицательное значение будет считано, но не прибавлено к общей сумме баллов (инструкция break находится перед инструкцией прибавления следующего значения). Поскольку после выполнения инструкции break переменная which не инкрементируется, я изменил ее начальное значение с 0 на 1. Таким образом, в результате мы получаем правильные значение для переменных total и which.

Обратите внимание, что внутри инструкции for нет условия окончания цикла. Но это не бесконечный цикл. Будьте уверены, в нужный момент инструкция break прервет выполнение цикла!

Можно поддаться искушению и использовать сигнальное значение внутри инструкции for. Например, следующий фрагмент программы на первый взгляд кажется правильным, поскольку задает цикл, который должен продолжаться, пока не будет введено отрицательное значение балла:

```
for(which=1; grade>=0; which++)
{
   grade=ask("Enter next grade:");
   total=total+grade;
}
```

Однако это не так. Вспомните, что условие проверяется перед началом итерации. Отрицательный балл будет считан, добавлен к общей сумме баллов, а значение переменной which инкрементировано. И только после этого проверяется условие выполнения цикла и, если оно ложно, цикл завершается! Более того, для первой итерации условие выполнения цикла вообще не определено (еще не введено ни одного значения балла).

Выбор варианта

Иногда может быть больше двух возможных вариантов развития событий. Например, если мы хотим, чтобы в соответствии со значением переменной code гимнаст занял одну из четырех позиций, можно написать следующий фрагмент кода:

```
if(code==1) Kim.ready();
if(code==2) Kim.up();
if(code==3) Kim.left();
if(code==4) Kim.right();
```

Для выбора варианта действий на основе значения одной переменной (ключа) используется инструкция switch (рис. 10.5).

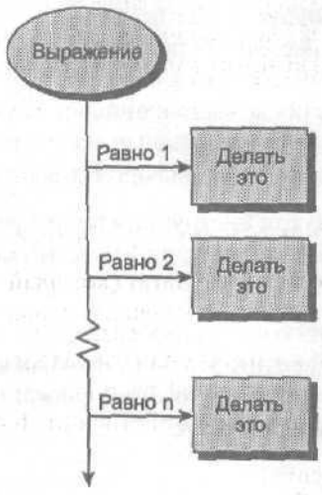


Рис. 10.5. *Алгоритм инструкции switch*

В нашем примере инструкция switch работает следующим образом:

Выбрать вариант из списка в соответствии со значением ключа:

Ключ равен 1:

Выполнить функцию ready()

Завершение инструкций данного ключа

Ключ равен 2:

Выполнить функцию ир()

Завершение инструкций данного ключа

Ключ равен 3:

Выполнить функцию left()

Завершение инструкций данного ключа

Ключ равен 4:

Выполнить функцию right()

Завершение инструкций данного ключа

Список вариантов исчерпан

В инструкцию switch входит заголовок и список ключей (его действие иллюстрирует рис. 10.5). Основной формат инструкции switch:

```
switch (выражение)
{
   список ключей
}
```

Под выражением понимается любое выражение, возвращающее целое значение (int, long или char), а список ключей (заключенный в фигурные скобки) состоит из любого числа ключей, каждый из которых имеет следующий формат:

```
case целое значение:
   инструкции
break;
```

Инструкция switch выполняется следующим образом:

- Вычисляется выражение, а его результат (который должен быть целым) поочередно сравнивается со значениями ключей (которые указаны после меток case).
- Если результат выполнения выражения совпадает со значением одного из ключей, выполняется список инструкций, расположенный следом за соответствующей меткой case, пока не встретится инструкция break.

Пример телефонной станции

Чтобы проиллюстрировать использование инструкции switch, рассмотрим «электронную телефонистку», которая должна благодарить абонента на его родном языке.

Можно написать программу, которая будет запрашивать телефонный код страны. Ввиду слишком большого числа возможных вариантов инструкция if неудобна, но зато как нельзя кстати инструкция switch. Если код страны сохранять в переменной country_code, то ее можно использовать в заголовке инструкции switch:

```
switch(country_code):
{
   case 1:
      message.say("Thank you");
   break;
   case 34:
      message.say("Gracias");
   break;
```

Чтобы усложнить программу и в то же время сделать ее интереснее, добавим туда Швейцарию, где даже в разных районах говорят на разных языках. Нет проблем! Просто кроме кода страны нужен еще код района. Соответствующая программа показана ниже.

```
ttinclude "franca.h"     //c4switch.cpp
void mainprog()
{
   Box display;
   int country, area;
   while(yesno("Shall we continue?"))
```

продолжение

```
  {
    case 55:          // Бразилия
    case 351:         // Португалия
      display.say("Obrihado");
    break;
    case 81:          // Япония
      display.say("Arigato");
    break;
    case 82:          // Корея
      display.say("Gamzahamnidah");
    break;
    case 52:          // Мексика
    case 34:          // Испания
    case 54:          // Аргентина
      display.say("Gracias");
    break;
    case 41:          // Швейцария
      switch(area)
      {
        case 21:      // Лузанна
        case 41:      // Люцерн
        case 22:      // Женева
          display.say("Merci");
          break;
        default:
          display.say("Danke");
      }
    break;
    case 49:          // Германия
    case 43:          // Австрия
      display.say("Danke");
    break;
    default:
      display.say("Thank you!");
    }
  }
}
```

Важно, чтобы вы усвоили следующие положения:

- Инструкция начинается с ключевого слова switch, за которым следует целая переменная (или выражение, результатом которого является целое значение), заключенная в круглые скобки. Это значение определяет действие, которое должно быть выполнено.
- Список ключей начинается и заканчивается фигурными скобками {}.

- Каждый ключ имеет определенное постоянное значение (константу), и список его инструкций заканчивается инструкцией break. Если вы опустите (или забудете) инструкцию break, то будет продолжаться выполнение инструкций следующего ключа, пока не встретится инструкция break. В нашем случае это даже удобно. Мы получаем возможность объединить инструкции для стран, в которых говорят на одном языке, и завершить их общей инструкцией break.
- Метка ключа состоит из ключевого слова case, константы и двоеточия. Первую инструкцию ключа можно размещать в той же строке, в которой находится метка. Например, допустима следующая инструкция:

```
case 51: message.say("obrigado");
```

- Если значение выражения в заголовке инструкции не совпадает ни с одним из перечисленных ключей, ни одной инструкции не выполняется.
- Можно задать ключ по умолчанию, который выбирается в том случае, когда значение выражения в заголовке инструкции не совпадает ни с одним из перечисленных ключей. Ключ по умолчанию обозначается ключевым словом default. Например:

```
default:
   message.say("Thank you");
```



ВНИМАНИЕ Убедитесь, что в конце инструкций каждого ключа вы не забыли вставить инструкцию break. В отличие от других конструкций, в которых для обозначения начала и конца составной инструкции используются фигурные скобки, в конструкции switch в качестве последней инструкции ключа требуется инструкция break. В противном случае программа будет продолжать выполнение инструкций, которые могут относиться уже к другим ключам,

Что нового мы узнали?

В этом уроке мы научились

0 Использовать инструкцию if.

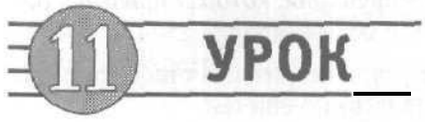
☑ Использовать инструкцию if...else.

0 Строить условные выражения.

0 Прерывать циклы.

0 Разрабатывать сложные циклы.

0 Использовать инструкцию switch.



Рекурсивные функции

- Рекурсивные алгоритмы
- Создание рекурсивных функций

Рекурсией (recursion) называется специальный технический прием, при котором решение проблемы происходит так, как будто упрощенная, но схожая проблема уже решена (хотя на самом деле нет). Кто же в таком случае решает упрощенную проблему? Здесь и выходит на сцену рекурсия — функция, которая призвана решать основную проблему, вызывает саму себя для решения упрощенной проблемы. Но что делать, если даже упрощенную проблему трудно решить? Ничего страшного, функция просто продолжает вызывать саму себя.

Пользуясь этим приемом, вы все время откладываете решение очередной проблемы, но при этом каждый вызов функции все более и более ее упрощает, пока проблема не становится тривиальной. Что делать теперь? Вы двигаетесь обратно и поочередно решаете отложенные проблемы.

На этом уроке нам будет помогать наш знакомый робот.

Рекурсивные алгоритмы

Рекурсивные алгоритмы для получения результата используют сами себя. Рассмотрим задачу определения числа шагов, которое нужно сделать роботу прежде, чем он упрется в стену. Рекурсивный ответ на этот вопрос выглядит так:

Расчет *расстояния*:

Если впереди стена, результат равен нулю

В противном случае:

Сделать шаг вперед

Результат равен *1+расстояние*



СОВЕТ

- Учитель, сколько шагов до вершины?
— Сделай один шаг и станешь на шаг ближе.

Полученное в результате выражение *1+расстояние* неизвестно, поскольку неизвестно *расстояние*. Мы откладываем вычисление текущего *расстояния* и переходим к рассмотрению нового. Попробуем понять, как работает данный алгоритм.

Когда вы собираетесь вычислять расстояние до стены, то первым делом проверяете, нет ли ее перед вами. В этом случае расстояние равно нулю. Однако, если стены впереди нет, мы делаем шаг вперед. Теперь результат равен единица плюс новое расстояние. Но как найти это новое расстояние? Прервите вычисление, вернитесь к началу алгоритма и снова проверьте, нет ли перед вами стены, и т. д.

В некий момент вы достигнете стены и результат станет равным *нулю*. Теперь можете, сделав шаг назад, возобновить прерванное на предыдущем этапе вычисление и добавить единицу к расстоянию от следующей ячейки (которое, как мы определили, равно нулю). В результате будет возвращен новый результат (единица), и, сделав еще один шаг назад, мы опять возобновим вычисления. Некоторые алгоритмы лучше всего можно выразить с помощью рекурсии.



ПРИМЕЧАНИЕ Расстояние от данной ячейки есть расстояние от следующей плюсединица.

Создание рекурсивных функций

Язык C++ поддерживает рекурсивные функции — функции, которые могут вызывать сами себя. Например:

```
int distance()
{
   if(Tracer.seewall()) return 0;
   Tracer.step();
   return 1+distance();
}
```

В этом примере реализован описанный выше рекурсивный алгоритм. Последняя инструкция возвращает значение выражения 1+distance(). Однако в этом выражении имеется вызов самой функции distance() (функция вызывает саму себя). Единственное отличие нового вызова от предыдущего в том, что робот стал на один шаг ближе к стене.

Сейчас вам было бы очень полезно представить себя роботом и воспользоваться приведенным алгоритмом, чтобы определить расстояние в шагах до ближайшей стены. Полезно было бы также, чтобы функция показывала ход вычислений. Например:

```
int distance()
{
   if(Tracer.seewall()) return 0;
   Tracer.step();
   Tracer.say(1+distance());
   return 1+distance();
}
```

Чтобы не вызывать функцию дважды, нужно сохранять результат в переменной:

```
int findwall()
{
```

```
    int steps;
    if(Tracer.seewall()) return 0;
    Tracer.step();
    steps=1+findwall();
    Tracer.say(steps);
    return steps;
}
```

Если вы выполните эту функцию, то увидите на экране следующее:

- Значения переменной steps будут равными 1, 2, 3...
- Эти значения появятся только после того, как робот достигнет стены. Почему?



ВНИМАНИЕ На первый взгляд кажется, что приведенная выше функция должна выполнять свое назначение, но это не так. Заметьте, что функция distanceQ вызывается дважды. Допустим, что вы робот. Будете ли вы идти к стене, чтобы вычислить расстояние и сообщить его, а затем возвращаться обратно, чтобы вернуть результат? Если функция distance() оставляет робота стоять возле стены, то такая функция будет работать неправильно, поскольку при повторном вызове distance() результатом будет единица. Попробуйте! Вы можете проверить это с помощью программы c4dist1.cpp.

Как работает рекурсивная функция?

При каждом вызове функции компьютер запоминает ту инструкцию в программе, в которой находится вызов функции, и затем переходит к выполнению функции. Создаются все переменные, которые объявлены в функции, а аргументы переписываются в параметры (если, конечно, они не передаются по ссылке). Когда функция вызывает саму себя, она как бы на время замораживается, и начинается выполнение новой копии этой функции.

Проиллюстрируем этот процесс на примере функции findwall(). Допустим, что при первом вызове функции робот находится на расстоянии трех ячеек от стены. Поскольку будут и другие вызовы, назовем этот вызов первым экземпляром функции.

Первый экземпляр

Когда функция вызывается в первый раз (другой программой), создается переменная steps и робот находится в ячейке номер 1. Поскольку стены впереди нет, функция заставляет его сделать шаг вперед. Теперь робот оказывается в ячейке номер 2.

Далее функция вычисляет значение переменной steps как результат выполнения выражения 1+findwall(). Здесь начинается рекурсия. Оставшиеся в функции инст-

рукции не выполняются. Функция замораживается вместе со всеми текущими переменными до тех пор, пока не будет вычислен результат выражения. Поскольку результата еще нет, никакого значения переменной steps не присваивается. Функция вызывается снова (запускается второй экземпляр функции).

Второй экземпляр

Запускается второй экземпляр функции. Создается еще одна переменная steps, и робот находится в ячейке номер 2. Поскольку стены впереди по-прежнему нет, функция заставляет его сделать шаг вперед. Теперь робот оказывается в ячейке номер 3.

Далее функция вычисляет значение переменной steps как результат выполнения выражения 1+findwall(). Снова возникает рекурсия. Оставшиеся инструкции не выполняются. (Помните, когда мы говорили о первом экземпляре функции, там происходили те же самые вещи.) Теперь замораживается этот экземпляр функции до тех пор, пока не будет вычислен результат выражения. Поскольку результата еще нет, никакого значения второй версии переменной steps не присваивается. Функция вызывается снова (запускается третий экземпляр функции).

Третий экземпляр

Запускается третий экземпляр функции. Создается еще одна переменная steps (уже третья версия), и робот находится в ячейке номер 3.

Поскольку теперь впереди стена, выполнение функции завершается и в качестве результата возвращается значение 0. Поскольку выполнение этого экземпляра функции завершилось и результат получен, может продолжиться выполнение предыдущей версии. Никакого значения переменной steps по-прежнему не присваивается (компьютер возвращается ко второму экземпляру функции).

Второй экземпляр

Компьютер продолжает вычисления. Второй версии переменной steps теперь присваивается значение 1+0, Робот отображает это значение на экране, и оно становится возвращаемым значением функции (которое здесь оказывается равным 1). Поскольку выполнение этого экземпляра функции завершилось и результат получен, может продолжиться выполнение предыдущей версии (наконец, компьютер возвращается к первому экземпляру функции).

Первый экземпляр

Компьютер продолжает вычисления. Исходной версии переменной steps теперь присваивается значение 1+1. Робот отображает это значение на экране, и оно становится возвращаемым значением функции (которое здесь оказывается равным 2). Выполнение первого экземпляра функции на этом завершается, и результат передается в исходную программу, вызвавшую эту функцию.

Совершенствование алгоритма

Кое что в этой функции мне не нравится. После измерения расстояния робот остается стоять у стены. Не лучше ли ему вернуться в исходное положение?

Ключевыми элементами функции являются: вычисление значения переменной steps и следующий при этом рекурсивный вызов функции findwaU(), поскольку компьютер должен выполнить следующее:

- Приостановить работу и запомнить все то, что может понадобиться для ее возобновления.
- Перезапустить функцию, чтобы продолжить вычисление расстояния.
- Возобновить отложенную работу при получении результата.

Если мы хотим вернуть робота в исходное положение, лучше, чтобы после проведения измерения он выполнил шаг назад. То есть мы как бы отменяем шаги после получения результата.

Для этого нам, возможно, подошла бы следующая функция:

```
int distance()
{
   int steps;
   if(Tracer.seewall()) return 0;
   Tracer.step();              // Шаг вперед
   steps=1+distance();         // Подсчет расстояния
   Tracer.say(steps);          // Вывод текущего расстояния
   Tracer.step();              // Шаг назад!
   return steps;
}
```

Кажется, мы забыли попросить робота развернуться. После всех ухищрений он по-прежнему движется к стене. Очевидно, его нужно развернуть. Но когда?



ВНИМАНИЕ Вы, наверное, думаете, что его нужно развернуть перед тем, как идти назад. Ни в коем случае! Не забудьте, что развернуться робот должен только однажды. Если же разворачивать его до выполнения шага назад, то он начнет вертеться. (Попробуйте! Это интересно!)

Лучше всего заставить его развернуться именно тогда, когда он встретит стену. В этом случае после выполнения измерения робот окажется повернутым в обратном направлении.

Окончательная версия функции выглядит следующим образом:

```
int distance()
{
   int steps;
   if(Tracer.seewall());
   {
   Tracer.right();
   Tracer.right();
   return 0;
   }
   Tracer.step();            // Шаг вперед
   steps=1+distance();       // Подсчет расстояния
   Tracer.say(steps);        // Вывод текущего расстояния
   Tracer.step();            // Шаг назад!
   return steps;
}
```

Более сложную версию этой функции можно найти в программе c4dist2.cpp прилагаемой дискеты.

Расходование памяти при вызовах рекурсивных функций

Рекурсивная функция запоминает значения всех своих переменных перед рекурсивным вызовом. Все объявленные в функции переменные в каждой новой версии создаются вновь, и при этом прежние версии продолжают храниться. Это также справедливо для всех аргументов, передаваемых по значению, которые, как и переменные, при каждом вызове функции создаются вновь и вновь. Вдобавок функция точно запоминает все, что она делала, чтобы впоследствии можно было возобновить работу.

Недостатки рекурсий

Рекурсивные алгоритмы хороши тем, что иногда могут предложить более простой путь решения проблем. Однако имеются и некоторые недостатки, о которых вы должны знать,

При каждом вызове функции:

- Создается область памяти для хранения всех объявленных в функции переменных и объектов. Так, в функции findwall() переменная steps объявлялась внутри функции.
- Аргументам и возвращаемым значениям функции также необходима некоторая область памяти.

- Компьютер должен сохранить все параметры вызова, в том числе и место в программе, с которого будет продолжена программа после завершения выполнения функции.

При каждом вызове функции резервируется некоторая область памяти. Если расстояние составляет 1000 шагов, то функция будет вызываться 1000 раз и 1000 областей памяти будет зарезервировано.

Нужно также учитывать, что каждая рекурсивная функция, помимо того, что занимает некоторую область памяти, еще и не слишком эффективно расходует процессорное время.

Завершение рекурсивных алгоритмов

Если понять концепцию, пользоваться рекурсивными алгоритмами и функциями очень легко. Главное удостовериться, что в какой-то момент рекурсии прекратятся. Продолжить отложенные вычисления можно только после того, как рекурсивная функция возвратит результат.

В наших примерах окончание рекурсий определить довольно просто: это момент, когда робот достигает стены. Но что будет, если никакой стены нет? Мы можем пользоваться этой программой только в том случае, если имеется гарантия, что стена будет найдена. В аналогичной ситуации, чтобы при исследовании лабиринта ограничить область поиска, мы либо запрашиваем вмешательства пользователя, либо задействуем таймер.

В любом случае убедитесь, что вы точно знаете условие завершения рекурсий и уверены, что оно будет выполнено. Не забывайте также (чтобы при случае этим воспользоваться), что все переменные или объекты, которые передаются по ссылке или определены глобально, повторно не создаются!

Третий этап великого похода

Поскольку мы теперь можем с помощью рекурсивной функции вычислять расстояние до следующей стены, кажется вполне логичным использовать рекурсивную функцию в программе c4seach.cpp, чтобы отмечать путь, по которому робот обходит лабиринт.

При определении расстояния до стены благодаря простому рекурсивному алгоритму мы гарантировали, что робот вернется в исходное положение. Естественно, что при исследовании лабиринта роботу понадобится иногда поворачиваться в ту или иную сторону. Можем ли мы в такой ситуации снова воспользоваться рекурсивной функцией, чтобы заставить его вернуться в исходное положение? Конечно! Все, что для этого нужно, — это воспроизвести все его шаги и повороты.

Начнем с небольшой модификации программы c4seach.cpp. В этой новой версии вместо цикла, в котором робот запрашивает разрешения вернуться, мы используем

рекурсивную функцию. Ниже представлена новая версия программы, программа c4backl.cpp:

```
#include "franca.h"
// Исследование лабиринта с помощью рекурсии
Robot Tracer(1);                    // c4back1.cpp
void explore1()
{
int derection;                      // Направление движения
if(yesno("Should I go on?"))        // Запрос о необходимости продолжать
{                                   // Если ответ yes, выполняются
   do                               // следующие инструкции:
   {
      direction=ask("which way should I go?");
      Tracer.face(direction);
      Tracer.mark(7);
   }                                // Повторять следующие инструкции,
                                    // пока впереди свободная ячейка:
   while(Tracer.seewall()); // Если следующая ячейка свободна, делаем следующее:
   Tracer.mark();                   // Закрашивание зеленым цветом
   Tracer.step();                   // Шаг вперед
   explore1();                      // Продолжение
                                    // Эта часть инструкций будет выполнена
                                    // только после окончания
                                    // исследования лабиринта!
   }
}
void mainprog()
{
   explore1();
}
```

Эта программа очень похожа на исходную. За исключением ряда незначительных изменений, основное отличие состоит в том, что в программе c4seach.cpp имелся цикл, условием завершения которого был наш ответ на запрос функции yesno(). В этой программе цикла на первый взгляд нет. Так ли это? Цикл есть, просто это рекурсивный цикл!

После того как на запрос функции yesno() вы ответите *yes,* робот переместится в новую ячейку и затем будет вызвана функция explore1(). Но этот вызов снова запускает функцию explorel(). Это ли не цикл? Если мы 15 раз отвечаем *yes,* значит, мы 15 раз вызываем функцию explore1(). При каждом вызове будет создаваться новая переменная distance, в которой будет сохраняться новое значение. Затем ро-

бот закрашивает следующую ячейку и перемещается в нее. В чем же отличие от цикла?

Отличие в том, что каждый вызов функции базируется на новом информационном материале, включая и значение переменной direction. Прежнее значение переменной direction при этом тоже сохраняется. Когда в конце концов вы ответите *no,* будет выполнен пятнадцатый экземпляр вызова функции, что даст возможность возобновить отложенное выполнение предыдущего вызова функции explore1() (для начала). После завершения выполнения 15 экземпляра содержимое последней версии переменной distance сбрасывается, и возобновляется выполнение 14 экземпляра. Такая последовательность действий будет продолжаться до тех пор, пока не завершится выполнение исходного экземпляра функции.

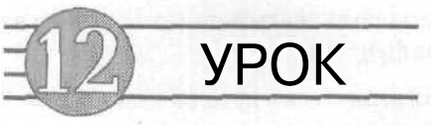
Можем ли мы в этой ситуации заставить робота вернуться в исходное положение? Аналогична ли эта ситуация с той, в которой с помощью рекурсивной функции мы измеряли расстояние до стены?

Конечно, да. Все что нам нужно — это воспроизвести в противоположном направлении шаг, сделанный в текущем экземпляре функции (как и в случае функции findwalLQ). Не забудьте, что теперь робот еще и поворачивается. Но мы знаем последнее направление движения робота, которое хранится в переменной direction. То есть если робот был направлен на север, мы сначала заставляем его повернуться на юг и только затем сделать шаг. Можете описать остальное? Пожалуйста!

Что нового мы узнали?

В этом уроке мы научились

- Идентифицировать проблемы, которые могут иметь рекурсивное решение.
- Понимать работу рекурсивного алгоритма.
- Пользоваться рекурсивными функциями.



Создание небольшого проекта

- Организация пользовательского интерфейса
- Перемещение робота
- Выбор функций
- Разработка простого приложения

На уроке 12 мы продолжим заниматься разработкой приложений. Мы будем работать над небольшим проектом, для которого потребуется разработка и компоновка программного обеспечения. В этом проекте снова будут использоваться объекты типа Robot,

Организация пользовательского интерфейса

В этом коротком проекте вы разработаете программу, которая сможет перемещать робота в любое место пустой комнаты. Положение робота будет определяться следующими координатами:

- Расстоянием в шагах (или ячейках) от левой стены (горизонтальная координата).
- Расстоянием в шагах (или ячейках) от правой стены (вертикальная координата).

Программное обеспечение предназначено для удовлетворения пользовательских запросов о требуемом положении робота. Поскольку в комнате нет внутренних стен, можно быть уверенным, что робот не окажется замурованным, если, конечно, его координаты останутся в допустимых пределах.

В качестве пользовательского интерфейса используется сам робот и несколько информационных рамок, отображающих его текущие координаты (горизонтальную и вертикальную). Рамки можно объявить следующим образом:

```
Box X("Horizontal:"), Y("Vertical:");
```

Общее описание движений робота

Робот всегда начинает движение из левой верхней ячейки. Поскольку эта ячейка находится в одном шаге от левой и верхней стены, естественно предположить, что эта ячейка имеет координаты (1,1).

Важно все время сохранять координаты робота. Если робот перемещается на один шаг вправо (на восток), то горизонтальная координата увеличивается на единицу. Что происходит, когда робот движется в других направлениях, можете определить сами.

В целом работу программного обеспечения можно описать следующим алгоритмом:

Повторять:

Получить новые значения координат

Переместить робота в ячейку с новыми координатами

Обновить текущие координаты

Нужно объяснять что-нибудь еще? Вы умеете создавать циклы и вероятно без проблем сможете организовать запрос на ввод новых координат. Действительно интересными проблемами являются перемещения робота и обновление координат.

Перемещения

Перемещение из одной позиции в другую выполняется очень просто. Допустим, робот находится в ячейке (1,1) и вам необходимо переместить его в ячейку (4,9). Что для этого нужно?

- По горизонтали переместиться из ячейки 1 в ячейку 4 — всего 3 шага.
- По вертикали переместиться из ячейки 1 в ячейку 9 ~ всего 8 шагов.

Таким образом, если робот находится в ячейке (x,y), необходимо переместить его в ячейку *(newx, newy):*

- По горизонтали его необходимо переместить на $newx-x$ шагов.
- По вертикали его необходимо переместить на $newy-y$ шагов.

Есть ли здесь проблемы? Серьезных нет — разве что вам придется иногда перемещаться на отрицательное число шагов, что означает двигаться на запад (по горизонтали) или на север (вертикали).

Выбор вспомогательных функций

Для решения проблемы разумно создать функцию. Допустим, у нас имеется функция, указывающая роботу, сколько шагов нужно сделать по горизонтали и вертикали. Эта функция проверит, будет число шагов положительным или отрицательным, и направит робота в соответствующем направлении:

```
void move(int xsteps, int ysteps)
```

Чтобы еще более упростить наш проект, представим другую функцию, которая реально будет перемещать робота из одной ячейки с заданными координатами в другую и обновлять текущие координаты:

```
void moveto(int &currentx, int &currenty, int newx, int newy)
```

Поскольку нам бы естественно хотелось, чтобы обновление координат выполнялось автоматически, передавать их функции лучше по ссылке. При наличии этих

функций основное тело цикла может быть записано в смешанной, программно-алгоритмической форме:

```
// Ввод новой горизонтальной координаты;
horiz=ask("horizontal:");
// Ввод новой вертикальной координаты:
vert=ask("vertical:");
// Перемещение в новое положение:
moveto(currentx, currenty, horiz, vert)
// Вывод координат на экран:
X.say(currentx);
Y.say(currenty);
```

Сам цикл можно легко контролировать с помощью функции yesno(), если она будет выдавать запрос о необходимости выполнить перемещение:

```
while (yesno("Want to move?"))
```

Теперь основная функция mainprog() почти готова. Осталось только объявить и инициализировать переменные.

Функция moveto()

Соответствующие коды для функций movetoQ и move() довольно интересны. Предполагается, что функция moveto() будет перемещать робота из ячейки с текущими координатами в ячейку с новыми координатами и обновлять значения текущих координат:

```
void moveto(int &currentx, int &currenty, int newx, int newy)
```

Вы уже, наверно, поняли, что в этой функции может использоваться функция move().

Как мы видели ранее, нужно переместить робота на newx–currentx шагов по горизонтали и на newy-currenty шагов по вертикали. То есть использовать функцию move() следующим образом:

```
move(newx-currentx, newy-currenty);
newx=currentx;
newy=currenty;
```

Не правда ли просто? Все трудности мы оставляем для следующей функции!

Функция move()

Теперь мы можем окончательно определить функцию move():

```
void move(int xsteps, int ysteps)
```

Эта функция будет определять, является значение переменной xsteps положительным или отрицательным, а затем перемещать объект в нужном направлении. Аналогичная процедура будет применяться к переменной ysteps.

Все, что необходимо сделать для горизонтальной координаты, — это повернуть робота на восток (положительное значение переменной x) или на запад (отрицательное значение переменной x) и сделать определенное число шагов. Затем то же самое необходимо сделать для вертикальной координаты.

```
void move(int x, int y)
{
   if(x>0)                          // Проверка горизонтального направления
   {
   Tracer.face(3);
   Tracer.step(x);
   }
   else
   {
      Tracer.face(1);               // Если значение x отрицательно,
      x=-x;                         // делаем его положительным
      if(x!=0) Tracer.step(x);
   }
   if(y>0)                          // Теперь делаем то же для y
   {
      Tracer.face(2);
      Tracer.step(y);
   }
   else
   {
      Tracer.face(0);
      y=-y;
      if(y!=0) Tracer.step(y);
   }
   Tracer.face(3);
}
```

Обратите внимание, что, во-первых, число шагов должно быть положительным, во-вторых, нельзя пройти ноль шагов.

Контроль ошибок

Чтобы пользователь не мог ввести неправильные координаты, необходимо обеспечить контроль ошибок. Ограничим допустимый диапазон координат значениями от 1 до 13.

Можно либо выдавать сообщение об ошибке, либо продолжать запрашивать введения правильных значений, В последнем случае решение имеет следующий вид:

```
do
   {
      horiz=ask("horizontal(between 1 and 13):");
   !
while((horiz<1)¦¦(horiz>13);
```

Завершение проекта

Окончательная версия проекта представлена в программе *c4move2.cpp*:

```
#include "franca.h"                    // c4move2.cpp
#include "robot.h"
Robot Tracer;
void move(int x,  int y)
{
   if(x>0)                             // Проверка горизонтального направления
   {
   Tracer.face(3);
   Tracer.step(x);
   }
   else
   {
      Tracer.face(1);                  // Если значение x отрицательно,
      x=-x;                            // делаем его положительным
      if(x!=0) Tracer.step(x);
   }
   if(y>0)                             // Теперь делаем то же для y
   {
      Tracer.face(2);
      Tracer.step(y);
   }
   else
   {
      Tracer.face(0);
      y=-y;
      if(y!=0) Tracer.step(y);
   }
   Tracer.face(3);
}
```

продолжение

```
void moveto(int &currentx, int &currenty, intx, int y)
{
   int stepsx, stepy;
   // Определение числа шагов по горизонтали
   stepx=x-currentx;
   // Определение числа шагов по вертикали
   stepy=y-currenty;
   move(stepsx, stepy);
   // Определение новых значений координат
   currentx =currentx+stepx;
   currenty= currenty+stepy;
}
void mainprog()
{
   int currentx=1, currenty=1;
   Box X("Horizontal:"), Y("Vertical:");
   int horiz, vert;
   for(; yesno("Want to move?");)
   {
      do
      {
         horiz=ask("horizontal(between 1 and 13):");
      }
      while((horiz<1) !; (horiz>13);
      moveto(currentx, currenty, horiz, vert);
      X.say(currentx);
      Y.say(currenty);
   }
}
```

Самостоятельная практика

О Модифицируйте программу торгового терминала, чтобы после ухода последнего покупателя она могла рассчитывать и отображать на экране следующие значения:

- Число покупок, суммарная стоимость которых меньше $10.00.
- Число покупок, суммарная стоимость которых находится в интервале от 10.00 до $50.00.

- Число покупок, суммарная стоимость которых находится в интервале от 50.00 до S 100,00.
- Числопокупок, суммарная стоимость которых превышает $100.00.

О Напишите программу для ввода с клавиатуры баллов успеваемости и вычисления среднего значения для студентов, набравших 50 и более баллов.



ПРИМЕЧАНИЕ Следующие три упражнения рассчитаны на читателей, имеющих математическую подготовку.

О Функция расчета факториала для неотрицательного числа п может быть определена рекурсивно:

- Если п равно нулю, факториал равен 1.
- Если п больше нуля, то факториал равен произведению п на факториал n-1.

О Напишите функцию вычисления факториала, использующую рекурсивное определение. Протестируйте функцию.

О Напишите функцию для расчета и возвращения абсолютного значения числа с плавающей точкой. Протестируйте функцию.



СОВЕТ Если переменная у положительна, ее абсолютное значение равно у. Если переменная у отрицательна, ее абсолютное значение равно -у.

О Напишите программу для ввода с клавиатуры баллов успеваемости (от 0 до 100). Эта программа должна определить, какое их количество лежит в интервале от О от 25, какое — в интервале от 26 до 50, какое — в интервале от 51 до 75 и, наконец, какое — в интервале от 76 от 100. Выведите результаты на экран.

Что нового мы узнали?

В этом уроке мы научились

0 Анализировать и выбирать пользовательский интерфейс.

0 Делать общее описание приложения.

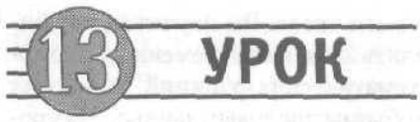
0 Выбирать функции и фрагменты программ, которые можно использовать в приложении.

0 Разрабатывать простое приложение.

Часть V

Числа

В части V вы продолжите знакомство с числовыми данными. Благодаря программному обеспечению книги вы сможете изучать операции с числами с помощью графики и анимации. Перед тем как работать с экранными объектами и анимацией, вы научитесь программировать достаточно сложные числовые выражения. Здесь вам понадобятся начальные знания геометрии.



Операции с числовыми выражениями

- Числовые типы данных в C++
- Создание и вычисление выражений
- Использование математических функций

На уроке 13 мы расширим наши познания в области чисел. Вы научитесь подбирать типы переменных для своих данных, вычислять значения переменных с помощью выражений, пользоваться библиотекой математических функций. Во многих приложениях требуется обрабатывать большие объемы числовых данных. На уроках 14 и 15 вам представится интересная возможность использовать приобретенные здесь навыки работы с числовыми выражениями.

Обработка чисел

Обрабатывать на компьютере числовую информацию приходится очень часто. Фактически, изначально компьютеры предназначались именно для этого. С их помощью вычислялись различные траектории, интегралы, решались уравнения и т. д, Компьютеры выдают информацию тоже в числовой форме, кроме того, числами могут кодироваться цвет, форма, звук и почти все, что можно себе представлять.

Поскольку широкое использование числовой информации требуется во многих приложениях, мы некоторое время потратим на изучение способов обработки числовых выражений. Чтобы сделать тему более интересной, на уроках 14 и 15 мы подключим к процессу обучения графику и анимацию.

Идентификация числовых типов данных

С числовыми типами данных и числовыми выражениями мы познакомились на уроке 5. Здесь мы разберем эту тему более подробно.

Компьютеры, как правило, не работают с числами, представленными произвольным числом цифр. Для компьютера гораздо проще в соответствии с *типом* (type) числа выделить область памяти заданного размера. Этот процесс аналогичен работе калькулятора. Если в нем только 8 разрядов, вы не сможете работать на нем с девятизначными числами.

Хранение целых чисел

Как вы уже знаете, компьютер может работать с целыми числами и числами с плавающей точкой. Для хранения целых чисел используются типы `int` и `long`. Ниже представлено их краткое описание:

- Мы уже использовали переменные типа int на предыдущем уроке. Для хранения целого числа (типа int) компьютер выделяет область памяти заданного размера.
  - Проверьте, в каком диапазоне значений могут храниться целые числа на вашем компиляторе, В большинстве компиляторов для хранения целых выделяется 16 разрядов. Таким образом, значения лежат в диапазоне от -32768 до +32767. Другие компиляторы, возможно, могут хранить числа большего размера.
- Тип Long описывает так называемые длинные целые и сообщает компилятору, что мы собираемся использовать целое значение с более широким диапазоном. В этом случае для хранения этого значения выделяется удвоенная, по сравнению с целым, область памяти, что позволяет обрабатывать большие числа.
  - Проверьте, в каком диапазоне значений могут храниться длинные целые на вашем компиляторе. В большинстве компиляторов для хранения длинных целых выделяется 32 разряда. Таким образом, значения лежат в диапазоне от -2147483648 до +2147483647. В любом случае целые типа long занимают вдвое большую область памяти и могут хранить гораздо большие числа, чем целые типа int



СОВЕТ Не старайтесь точно запоминать эти числа — просто помните, что это примерно по два миллиарда в каждую сторону.

Беззнаковые целые

Если вы имеете дело с переменной, принимающей только положительные значения (например, число студентов в классе), то можете увеличить диапазон значений типа int (или типа long), поставив перед объявлением переменной ключевое слово unsigned. Например:

```
unsigned int число_студентов;
```

Поскольку у вас нет необходимости в отрицательных значениях, компьютер сможет хранить положительное число вдвое большего размера, чем раньше. Если диапазон значений вашего компилятора для хранения данных типа int от -32768 до +32767, то для беззнаковых целых он составит от 0 до +65535.



ПРИМЕЧАНИЕ Лучше без необходимости не злоупотреблять беззнаковыми целыми. Для числа студентов в классе наверняка хватит 32767 целых чисел.

Короткие целые

Если для вас важна экономия памяти, можете объявлять переменные, предназначенные для хранения небольших целых значений, ставя перед их объявлением ключевое слово short. Например:

```
short int color;
```



ПРИМЕЧАНИЕ Проверьте диапазон значений вашего компилятора для хранения коротких целых. В большинстве компиляторов для них выделяется 8 разрядов, что дает диапазон значений от −128 до +127. Вы можете объединить данные типа unsigned и short.

Хранение чисел с плавающей точкой

Существуют задачи, для решения которых целые числа не подходят. Например, часто приходится использовать десятичные и/или натуральные дроби.

- Стандартным типом данных для хранения чисел с плавающей точкой является тип float. Данные этого типа занимают несколько большую область памяти, чем данные типа int, и, поскольку организация их хранения сложнее, для обработки чисел с плавающей точкой требуется больше времени, чем для обработки целых. Обычно числа с плавающей точкой лежат в диапазоне от $3{,}4\times10^{-38}$ до $3{,}4\times10^{38}$ и могут быть как положительными, так и отрицательными, а также содержать до семи значащих цифр.



ПРИМЕЧАНИЕ Проверьте диапазон значений, соответствующий вашему компилятору. У разных компиляторов он может быть разным.



ПРИМЕЧАНИЕ Если вы попытаетесь поместить в память число 1.0000001, то компьютер не сохранит его полностью. Будет сохранено только семь основных значащих цифр (1.000000). Положение десятичной точки при этом никакой роли не играет. Например, число 234.56789 будет урезано до 234,5678.

- Другим типом данных для работы с большими числами является тип double. Данные этого типа аналогичны данным типа float, но предназначены для хранения значений с большим числом значащих цифр. Точный диапазон значений зависит от компилятора.

- Если вам необходимо обрабатывать числа с еще большим числом значащих цифр, то при объявлении переменных перед ключевым словом double необходимо еще ключевое слово Long. Точный диапазон значений также зависит от компилятора.

Константы

Кроме целых чисел и чисел с плавающей точкой, нам могут понадобиться целые константы и константы с плавающей точкой.

Константами называются значения, которые нельзя изменять в процессе выполнения программы. Имеются три варианта констант:

- *Буквальные константы* (literal constants) — воспроизводят значение буквально.
- *Именованные константы* (labeled constants) — выглядят в точности как переменные, которым назначено имя и присвоено значение. Но в отличие от переменных вы указываете компилятору, что это значение должно оставаться неизменным.
- *Перечислимые константы* (enumerated constants) — представляют собой ряд именованных констант, которым вы можете присваивать специальный тип и которые используют список имен для обозначения значений.

Целые константы

Целые константы очень похожи на целые числа, с которыми мы работаем каждый день. Отметим только некоторые детали:

- Перед числом можно ставить знак + или -.
- Нельзя использовать десятичные запятые или точки.
- Нельзя вставлять пробелы между цифрами.
- Нужно стараться не ставить перед числами незначащие нули, которые могут только запутать программу.



ВНИМАНИЕ Избегайте чисел, начинающихся с нуля. Эти числа зарезервированы в C++ для обозначения целых констант в восьмеричной системе счисления. Запись чисел в этой системе отличается от обычной записи. Например, в C++ константа 010 означает восьмеричную запись десятичного числа 8.

Константы с плавающей точкой

В C++ допустимо два формата записи констант с плавающей точкой. Первый формат аналогичен нашей обычной нотации:

- Перед числом можно ставить знак + или -.

- Можно ставить одну (и только одну) десятичную точку.
- Нельзя вставлять запятые или пробелы между цифрами.

Примеры констант с плавающей точкой:

35.

73.12

0.001

+54.3

–0.2

Вторым форматом является экспоненциальная нотация, аналогичная представлению больших величин в физике. Эта нотация состоит из двух частей: мантиссы, представленной числом с плавающей точкой (как в приведенных выше примерах), и показателя, состоящего из буквы E, за которой следует целое число, представляющее показатель.

Например, рассмотрим следующее число:

6.02×10^{23}

В C++ его можно записать, как 6.02E23 или 6.02E+23. Обе части — мантисса и показатель — могут иметь знаки.

Именованные константы

Для обозначения значения константы допустимо указывать имя. Если вы собираетесь в своей программе несколько раз использовать число 3.1416, то удобнее обозначить его следующим образом:

```
float pi=3.1416;
```

В этом случае все значения 3.1416, которые имеются в программе, заменяются переменной pi. Единственная проблема может случиться, если при выполнении программы вы случайно попытаетесь изменить значение переменной pi. Чтобы этого не случилось, включите в объявление ключевое слово const. Тогда это объявление будет выглядеть так:

```
const float pi=3.1416;
```

В этом случае компьютер не допустит изменения значения pi. Любая инструкция, в которой будет предпринята такая попытка, приведет к сообщению об ошибке.

Именованные константы удобны еще и тем, что при необходимости позволяют легко изменить значение константы. Например, вы можете решить вместо значения 3.1416 использовать более точное значение 3.14159. Гораздо проще изменить одно объявление, вместо того чтобы просматривать всю программу в поисках инструкций, в которых встречается значение 3.1416! Ключевое слово const позволяет

включать в программу переменные, значение которых в процессе работы программы должно оставаться неизменным.

Перечислимые константы

Перечислимые константы особенно полезны, когда те или иные значения кодируются последовательностями целых чисел. Возьмем, к примеру, направления на север, восток, юг и запад, которые можно закодировать числами 0, 1, 2 и 3. Чтобы отказаться от использования чисел, можно сделать следующее объявление:

```
int N=0, E=1, S=2, W=3;
```

Или лучше, как мы теперь знаем:

```
const int N=0, E=1, S=2, W=3;
```

В любом случае вместо числового кода программа будет работать с идентификаторами N, E, S и W. Например:

```
if (direction==N) ...
```

Тот же результат, но с дополнительными преимуществами может быть достигнут с помощью объявления перечислимых констант:

```
enum тип {список_имен}
```

Или в нашем случае:

```
enum directions{N, E, S, W};
```

В этой инструкции идентификаторы, перечисленные в списке, объявляются константами типа int, и им по умолчанию присваиваются значения 0, 1, 2 и 3. Если ничего другого не задано, то первый идентификатор инициализируется нулем, а каждый следующий — следующим значением целого.

В этом объявлении тип указывать не обязательно, поэтому в нашем случае в принципе можно опустить слово directions (направления). Тем не менее задание типа удобно, поскольку дает возможность объявить в программе переменную этого нового типа. Например, в следующей инструкции объявляются две новые переменные, которым в дальнейшем можно присвоить значения любых нужных нам направлений:

```
directions comingfrom, goingto;
```

Таким образом, допустимыми становятся следующие инструкции:

```
comingfrom = N;
goingto = S;
```

Однако, если вы попытаетесь присвоить этим новым переменным значение другого типа (пусть даже целое), компилятор выдаст вам соответствующее предупреждение. Например, недопустимо использование в программе следующих двух инструкций:

```
comingfrom = 8;
goingto = 2;
```

И наконец, последовательность идентификаторов можно объявлять не только по умолчанию, но и по отдельности присваивать нужное значение каждому идентификатору. Например:

```
enum colors (red=1, green, pink=5, yellow);
```

В этой инструкции идентификатору red присваивается значение 1, идентификатору green — значение 2 (то есть по умолчанию предыдущее значение плюс единица), идентификатору pink — значение 5 и идентификатору yellow — значение 6 (по умолчанию).

Выражения

В этом разделе на нескольких примерах рассматриваются правила выполнения арифметических операций и присваивания значений.

Комбинирование типов, переменных и констант

В выражениях можно объединять переменные и константы типов int и float. Так, в выражении из следующего примера сначала из значения переменной x вычитается значение переменной i, а затем прибавляется 1. Результат сохраняется в переменной y:

```
float x=0.5, y;
int i=5;
y = x - i + 1;
```

До сих пор все было так, как мы себе и представляли. В самом деле, способ представления арифметических выражений в C++ (как и в большинстве других языков программирования) очень похож на обычную математическую запись. Перед тем как рассматривать какие-либо правила, отметим следующие положения:

- В математике допускается отсутствие оператора умножения: xy означает значение x, умноженное на значение y.
  - В C++ такая запись недопустима. Компилятор решит, что это не x, умноженное на y, а объект с именем xy. Следовательно, необходимо добавлять оператор умножения, который обозначается символом * (звездочка).
- В математике горизонтальная черта, сверху и снизу от которой находятся числа, означает частное этих значений.
  - Поскольку программа вводится с клавиатуры, то в ней такая запись трудно осуществима. Вместо этого для обозначения деления используется косая черта /. При этом делимое и делитель остаются на одной строке. Например, x/y означает, что значение переменной x делится на значение переменной y.

О В математике при выполнении выражения целые числа и числа с плавающей точкой обычно не различаются.

- В большинстве языков программирования целые числа и числа с плавающей точкой трактуются совершенно по-разному. Так, результат от деления двух целых чисел также является целым! Например, частное выражения 3./2. (заметьте, оба числа с плавающей точкой) равно 1.5, а частное выражения 3/2 (оба числа целые) равно 1, поскольку учитывается только целая часть результата. Схожие проблемы возникают и при выполнении других операций.

Например, результат выполнения выражения следующего примера, вероятнее всего, будет неправильным, поскольку может превысить допустимый для вашего компилятора диапазон хранения целых чисел:

```
int i, j;
i=30000;
j = i + i;
```

Значение 60 000 значительно превышает допустимый для хранения целых диапазон значений.

Теперь, когда вы познакомились с этими отличиями, можно перейти к правилам.

Арифметические операторы

В выражениях допустимы следующие арифметические операторы:

+	сложение
–	вычитание
*	умножение
/	деление
%	остаток от деления

Кроме того, для присваивания переменной результата выполнения выражения может использоваться оператор присваивания (=). Например:

```
x=3*a+1; // Умножаем а на 3, прибавляем 1 и сохраняем результат в х
y=a+1*3; // Умножаем 1 на 3, прибавляем к а и сохраняем результат в у
```

Второй пример иллюстрирует, что в выражениях, написанных на C++, как и в математике, умножение обладает приоритетом над сложением и выполняется первым. Вы наверное помните, что выражение x+3a означает, что сначала вы должны умножить а на 3, а затем прибавить результат к х.

Правила приоритета:

О Высший приоритет имеют операторы *, / и %.

О Низший приоритет имеют операторы + и –.

О Операторы, имеющие высший приоритет, выполняются первыми.

- Если несколько операторов имеют одинаковый приоритет, они выполняются поочередно слева направо.

С помощью скобок можно менять порядок выполнения операций:

```
3*(x+1) // Прибавить x к 1, а затем умножить на 3
3*x+1   // Умножить x на 3, а затем прибавить 1
```

При необходимости одни скобки можно ставить внутрь других. В этом случае операция во внутренних скобках выполняется первой. Так, в следующем примере сначала складывается 1 и y, затем x делится на результат сложения, к новому результату прибавляется 4 и наконец последний результат умножается на 3:

```
3*(x/(y+1)+4);
```

Обратите внимание, что x делится на y+1, а не на y+1+4.

Присваивание

При выполнении операции присваивания результат выполнения выражения, стоящего справа от оператора присваивания (=), присваивается переменной, стоящей слева от оператора присваивания. Помните, что оператор присваивания означает совсем не то, что знак равенства в математике! Например, первая из следующих двух инструкций в C++ допустима, а вторая нет:

```
s=s+3;
s+3=s;   // Ошибка!
```

Генерирование результатов

Помимо передачи значения из одного места в другое, оператор присваивания генерирует результат, который и является передаваемым значением. Другими словами, следующая инструкция, кроме того, что сохраняет в переменной s значение 3, еще генерирует значение 3 в качестве результата:

```
s=3;
```

Именно поэтому с точки зрения синтаксиса компилятор считает правильной следующую инструкцию:

```
if(s=3)...
```

В этой инструкции значение переменной s не сравнивается с числом 3, а число 3 сохраняется в переменной s и генерируется в качестве результата выражения. А поскольку 3 не равно нулю, результатом выражения цикла if будет *истина*.

Множественные присваивания

Поскольку каждое присваивание генерирует результат, можно написать следующую инструкцию: -

```
x=y=1;
```

Здесь в операции присваивания y=1 генерируется результат 1, который и используется в следующем операторе присваивания.



ПРИМЕЧАНИЕ Множественные присваивания выполняются справа налево.

Преобразования

Если выражение справа от оператора присваивания и переменная слева относятся к разным типам данных, то тип результата выполнения выражения преобразуется (convert) в тип переменной, стоящей слева от оператора присваивания. Рассмотрим следующий пример:

```
int m=1, j=2, k;
float x=0.5, y;
m=x;
```

Результатом выполнения выражения будет число 0.5, которое преобразуется в значение типа int, а затем сохраняется в переменной т. Из-за преобразования новое значение переменной т будет равно нулю. Рассмотрим теперь следующее выражение:

```
x=m=x;
```

Результатом выполнения этого выражения будет ноль в обеих переменных (как в m, так и в x).

Рассмотрим еще одно выражение:

```
m=x=x;
```

Какой будет результат, как вы думаете?

Смешивание разных типов данных

Выражения могут содержать объекты разных типов. Когда в операцию вовлечены операнды разных типов, компилятор пытается повысить (promote) один из них. Например, если значение типа float прибавляется к значению типа int, то сначала тип int повышается до типа float и только потом выполняется операция.

Повышения и понижения типа

Когда выполнение выражения ведет к операции между целым числом и числом с плавающей точкой, то целое сначала преобразуется в свой эквивалент с плавающей точкой и только потом выполняется операция (в этом случае мы говорим, что тип int был повышен до типа float). Аналогично, если значение типа float должно присваиваться переменной типа int, то компилятор понижает (demote) значение типа float, отсекая дробную часть, и рассматривает результат как целое.

Этими преобразованиями типов можно управлять. Для преобразования переменной или выражения в другой тип нужный тип данных указывается в скобках непосредственно перед переменной или выражением.

Например, выражение x=(float)m/(float)j; гарантирует, что перед делением переменных m и j они будут преобразованы в тип float[1].

Возвращаемые значения функций

Помимо переменных (объектов) и констант, в выражения могут входить функции. Любая функция может генерировать результат любого типа, который, в свою очередь, может использоваться как часть выражения. Чтобы функция генерировала результат, необходимо сделать следующее:

- Задать тип результата. Например, результат может быть типа int, float, long и т. д. Тип void означает отсутствие возвращаемого результата.
- Обозначить возвращаемый результат с помощью инструкции return,

Инструкция return

Инструкция return состоит из ключевого слова return, за которым записывается выражение. Значение выражения возвращается в качестве результата. После выполнения инструкции return функция завершается: следующая инструкция функции (если она есть) уже не выполняется.

Пусть, например, требуется создать функцию для определения расстояния, которое пролетает тело в свободном падении. Соответствующая физическая формула имеет вид:

$h=0.5gt^2$

Здесь h — это высота, g — ускорение свободного падения и t — время. Поскольку вблизи поверхности Земли значение g можно считать константой (9.81 метров на секунду в квадрате), то для заданного значения t можно рассчитать значение h, которое и станет результатом выполнения функции. Каков будет его тип? Разумнее всего выбрать тип float, поскольку расстояние, выраженное в метрах, может быть дробным.

Если назвать эту функцию height, то можно составить следующий код:

```
float height(float t)
{
   return 0.5*g*t*t;
}
```

Можно ли заменить 0.5 на 1/2? Ни в коем случае! Если вы напишете 1/2, это подразумевает частное от деления одного значения (1) типа int на другое значение (2) типа int. Таким образом, в результате должно получиться третье значение типа int, что означает результат, равный нулю. Избежать этого можно, если изменить тип одного или двух операндов на тип float. Например, подойдет выражение 1./2. Однако тогда, каждый раз вычисляя это выражение, компьютер будет снова делить 1 на 2. Поэтому лучше все-таки записать 0.5.

[1] Описанные здесь операции повышения, понижения и преобразования типа обычно называют операциями приведения типа (cast). — *Примеч. ред.*

В C++ нет оператора для вычисления квадрата числа. Самое простое решение этой проблемы основано на том, что квадрат t есть произведение t на t Можно также использовать стандартную математическую функцию sqr(t), как показано в следующем разделе.

Библиотека math.h

Ряд наиболее популярных математических функций включен в заголовочный файл math.h. Здесь вы познакомитесь только с самыми важными из них.

В качестве аргументов эти функции получают значения типа double и возвращают результат того же типа. Поскольку преобразование типа выполняется автоматически, можно использовать аргументы типа float, а также присваивать тип float результату.

Математические функции представлены в табл. 13.1.

Таблица 13. i. Основные функции библиотеки math.h

Функция	Назначение
sin(x)	Возвращает синус x
cos(x)	Возвращает косинус x
tan(x)	Возвращает тангенс x
asin(x)	Возвращает арктангенс x
acos(x)	Возвращает арккосинус x
atan(x)	Возвращает арктангенс x
exp(x)	Возвращает экспоненту x
Iод(x)	Возвращает натуральный (по основанию e) логарифм x
Iog10(x)	Возвращает десятичный логарифм x
sqrt(x)	Возвращает квадратный корень x
sqr(x)	Возвращает квадрат x
fabs(x)	Возвращает абсолютное значение числа x
ceiI(x)	Возвращает ближайшее целое, большее x
floor(x)	Возвращает ближайшее целое, меньшее x
rand()	Возвращает случайное целое (диапазон значений зависит от компилятора)

Пример

Приведенная ниже программа предназначена для вычисления квадратного корня значения, которое вводится с клавиатуры.

```
ttinclude "franca.h"
#include <math.h>
voidraainprogO
{
   float value, root;
   // Запрос на ввод положительного значения:
   value=ask("Enter a positive value:");
   if(value>=0)
   {
      root=sqrt(value);
      Cout<<root;
   }
   else
   {
   // Сообщение об ошибочном вводе отрицательного значения
   Cout<<"Sorry, negative value!";
   }
}
```

Что нового мы узнали?

В этом уроке мы научились

- **Выбирать типы переменных для обработки данных.**
- **Получать результаты выполнения выражений.**
- **Использовать функции из библиотеки math.h.**

14 УРОК Работа с графикой

- Координаты точек на экране компьютера
- Экранные объекты
- Перемещение экранных объектов
- Смена систем координат

На уроке 14 мы будем работать с графикой, поскольку она является интересным приложением обработки чисел и ее значение в программировании постоянно растет,

Из-за сложности графических объектов новичкам обычно не удается научиться с ними работать, но благодаря программному обеспечению книги для вас это не составит труда. Мы сможем работать с окружностями, квадратами, прямоугольниками. Будет использован специальный класс объектов — экранные объекты, которые помогут нам еще лучше изучить обработку чисел и графики.



ПРИМЕЧАНИЕ Объектов типа Cirlce, Square, Stage и других нет в стандартном языке C++. С ними можно работать только при использовании программного обеспечения книги.

Графика

Одним из наиболее интересных приложений числовых вычислений является работа с графическими объектами на экране компьютера. Манипуляции с графикой будут преследовать две цели:

- Вы научитесь (на практике) обращению с графическими объектами, что очень интересно само по себе.
- Вы освоите (на практике) некоторые аспекты числовых вычислений.

Размещение точек на экране

Чтобы понять, как работают графические объекты, для начала необходимо понять, как расположены точки на экране компьютера. Поскольку экран представляет собой двумерный объект, то положение точки на нем задается координатами *x* и *y*.

В сущности, это та же система координат, которую мы использовали в коротком проекте урока 12, чтобы размещать и находить в комнате робота.

Начало координат на экране компьютера находится в его левом верхнем углу — по определению это точка (0,0). Координата *x* возрастает слева направо (как и принято в геометрии), однако координата *y* — сверху вниз (противоположно тому, как принято в геометрии).

Координатные единицы

На экране компьютера может быть представлено только конечное число точек. И не только потому, что экран имеет конечные размеры, но еще и потому, что существует некоторое минимально возможное расстояние между точками. Реальное число точек зависит от типа вашей видеокарты. Например, если на вашем компьютере экранное разрешение 1024x768, то ваш экран содержит 1024 точки по горизонтали и 768 по вертикали. В зависимости от размера экрана компьютер будет работать с одним из заданных разрешений. На рис. 14.1 показаны наиболее распространенные из них.

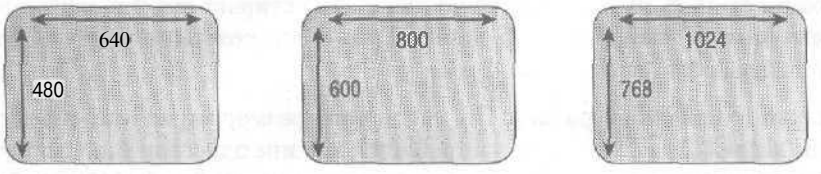


Рис. 14.1. *Типичные экранные разрешения*

Поскольку число точек на любом отрезке экрана всегда конечно, то разумно в качестве координат заданной точки использовать минимальное число точек, которые можно отложить от нее до начала координат по горизонтали и вертикали. Минимальной единицей такой системы координат является так называемый *пиксель* (pixel).

ПРИМЕЧАНИЕ Пиксель определяет элемент изображения — точку, которая может быть представлена на экране заданным цветом.

Очевидно, что координаты можно переопределить так, чтобы вместо пикселей можно было пользоваться дюймами или сантиметрами. Как это сделать, мы узнаем несколько позже.

В пиксельной системе координат координаты точки определяются расстояниями (по горизонтали и вертикали) в пикселях от начала координат в левом верхнем углу экрана.

Не забывайте, что в отличие от геометрии вертикальная координата возрастает сверху вниз.

Изображение экранных объектов

Для изучения графических приложений я приготовил класс экранных объектов. Этот класс включен в заголовочный файл franca.h. Вот некоторые из них:

- Информационные рамки (которые мы используем начиная с урока 2).
- Квадраты.
- Окружности.

Положение квадратов и окружностей на экране определяется координатами их центров, а положение рамок — координатами их левого верхнего угла.

Над экранными объектами можно выполнять следующие операции:

- Можно размещать их в любом месте экрана. Функция place(x, y) размещает объект в точке с координатами *(x, y)*. Например, ball.place(20,20);.
- Можно отображать их на экране (вообще говоря, объекты автоматически не отображаются). Функция show() рисует объект в текущей точке экрана. Например, ball.show();.
- Можно стирать их с экрана. Функция erase() стирает объект, закрашивая его белым цветом. Например, ball.erase();. Будьте осторожны, не сотрите объект перед тем, как его перемещать.
- Можно изменять их размер. По умолчанию размер экранного объекта равен 20 пикселей. Для его изменения требуется задание одного или двух аргументов. При задании только одного аргумента — например, ball.resize(40); — оба размера по горизонтали и вертикали становятся равными этому единственному аргументу. В противном случае ширина и высота объекта будут отличаться. Например, вызов ball.resize(40, 30); преобразует окружность в эллипс высотой 40 и шириной 30 пикселей. Для изменения размера имеются две функции resize() и absize(). Размер, задаваемый функцией resize() (в отличие от absize()), зависит от заданного масштаба.
- Можно изменять их цвет. Для этого используется функция color(). По умолчанию экранные объекты при создании внутри закрашены белым, а по контуру черным цветом. Задавая один цвет, объект можно закрасить этим цветом как внутри, так и по контуру, например ball.color(2);. Задавая два цвета, объект и его контур можно закрасить по отдельности, например ball.color(2, 4);.

Над экранными объектами типа Box можно выполнять некоторые специальные операции:

- Внутри информационной рамки можно вывести сообщение. Например, если объявить переменную message типа Box, то инструкция message.say(«Here!»); приведет к появлению внутри рамки сообщения *«Here!»*. В качестве аргумента можно также использовать целое число или число с плавающей точкой.
- Каждая информационная рамка может быть снабжена меткой, идентифицирующей сообщения. Внутри рамки метки располагаются над сообщениями. Записанные одна за другой инструкции message.label(«Your change:»); и message.say(change); приведут к появлению внутри рамки метки и сообщения. Например, если значение переменной change равно 12.45, то содержимое этой рамки будет выглядеть так:

Your change:

12.45

Поведение информационных рамок несколько отличается от поведения других экранных объектов, поскольку они появляются автоматически, когда вы использу-

ете функцию say(). Другими словами, рамки не требуется выводить на экран. Кроме того, координаты рамки относятся к ее левому верхнему углу, а не к центру.

В табл. 14.1 и 14.2 обобщены функции управления экранными объектами.

Таблица 14.1. *Функции управления экранными объектами*

Сообщение	Назначение	Аргументы
place(float float)	Размещение объекта	Координаты *x, y*
show()	Вывод объекта на экран	Нет
erase()	Стирание объекта с экрана	Нет
resize (float)	Изменение размера объекта	Новый размер
absize(float)	Изменение абсолютного размера объекта	Новый размер
color(int, int)	Изменение цвета объекта	Цвета
scale (float float)	Изменение масштабов	Новые масштабы
origin(float, float)	Изменение начала отсчета	Координаты *x, y*

Таблица 14.2. *Функции управления экранными объектами типа Box*

Сообщение	Назначение	Аргументы
label(«Label»)	Вывод метки	Строка символов, идентификатор строки символов или целое
say(«Sentence»)	Вывод сообщения	Строка символов, идентификатор строки символов, целое или число с плавающей точкой
resize(float)	Изменение размера объекта	Новый размер

Поскольку информационные рамки также являются экранными объектами, к ним можно применять функции place(), show() и erase().

При объявлении объекта Circle по умолчанию предполагается, что диаметр этой окружности равен 20 пикселям. Затем этот размер можно изменить. Например, в следующем фрагменте создается окружность диаметром 10 пикселей:

```
Circle mycircle;
mycircle. resize(10);
```

Аналогично, функция Square() создает квадрат со стороной, по умолчанию равной 20 пикселям (как и диаметр окружности).

Цвета

Экранные объекты можно закрашивать разными цветами. В объекте могут присутствовать два цвета — внутренний и цвет контура. Например, черная окружность может быть внутри красной. Цвета кодируются следующими цифрами:

0 белый

1 красный

2 светло-зеленый

3 голубой

4 светло-синий

5 розовый

6 желтый

7 черный

Как уже говорилось, в ваших программах может оказаться полезной следующая инструкция:

```
enum(white, red, green, blue, lightblue, pink, yellow, black);
```

Если значение кода задать большим 7, будет использован остаток от его деления на 7.

Умолчания

При создании экранных объектов по умолчанию происходит следующее:

- Информационные рамки располагаются вертикально друг под другом в правой части экрана; остальные объекты размещаются в точке с координатами (0, 0).
- Все объекты создаются белыми с черным контуром.

Использование целых значений

Несмотря на то что координаты задаются в пикселях, экранные объекты поддерживают задание координат и размеров в виде чисел с плавающей точкой. Даже при задании целых компилятор автоматически преобразует их в значения с плавающей точкой. Очень важно научиться правильно использовать значения типа float, поскольку в дальнейшем они пригодятся нам для изменения масштаба.

Следующий фрагмент программного кода создает окружность с центром в точке (50, 50) и отображает ее на экране:

```
tfinclude "franca.h"
void mainprog()
{
   int x, y;
   x=50;
   y=50;
   Circle mycircle;
   mycircle.place(x, y);
   mycircle.show()
}
```

Не забывайте, что все используемые в программе экранные объекты (как и объекты любых других классов) должны быть объявлены.

Программа c5circ1.cpp, являющаяся модификацией предыдущего фрагмента, позволяет запрашивать координаты окружностей и размещает их значения в информационных рамках. Если выполнить эту программу, введя координаты (50, 100), то экран вашего компьютера будет выглядеть так, как показано на рис. 14.2.

```
#include "franca.h"
                    c5circ1.cpp
// Программа рисования окружностей,
// положение которых определяется пользователем:
void mainprog()
{
   Box coordx("X:"), coordy("Y:"); // Метки информационных рамок
                                   // для ввода координат x и y
   int x, y;                       // Объявление координат
   Circle mycircle;                // Объявление окружности
   do
   {
      // Запрос значений координат:
      x=ask("Enter x coordinate:");
      y=ask("Enter y coordinate:");
      mycircle.place(x, y);        // Размещение окружности
      mycircle.show();             // Вывод окружности на экран
      // Вывод в информационных рамках значений координат:
      coordx.say(x);
      coordy.say(y);
   }
   // Запрос о необходимости следующей попытки:
   while(yesno("Wanna try again?"));
}
```

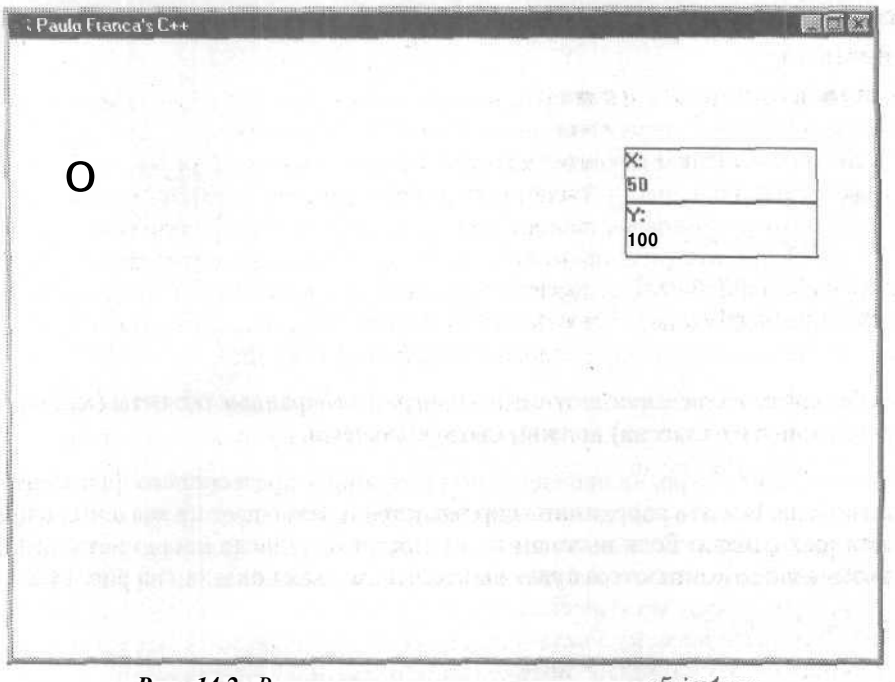


Рис. 14,2. *Результаты выполнения программы c5circ1.cpp*

Обратите внимание на объявление двух новых объектов coordx и coordy типа Box. В информационных рамках coordx и coordy отображаются значения координат центра окружности соответственно x и y. Хотя эти рамки можно разместить в любом месте экрана, они были автоматически выведены в точку, заданную по умолчанию.

В следующих строках запрашиваются значения координат x и y:

```
x=ask("Enter x coordinate:");
y=ask("Enter y coordinate:");
```

Это позволяет вам разместить окружность в любом месте экрана, а не в точке с координатами (50, 50), как было показано на рис. 14.2. Основная часть программного кода размещена в цикле do...while:

```
do
{
...
}
while(yesno("Wanna try again?"));
```

Что делает этот цикл? Итерации продолжаются до тех пор, пока условие истинно. Таким образом, если на запрос функции yesno() отвечать *yes*, программа будет продолжать запрашивать новые координаты и рисовать окружности. Для завершения цикла нужно ответить *no*.

Удаление экранных объектов

Когда мы рисуем новую окружность, прежняя остается на месте. А если мы не хотим ее больше видеть? Для удаления объектов предназначена функция erase(). Чтобы ее использовать, в подходящее место программы нужно включить следующую инструкцию:

```
mycircle.erase();
```

Что я понимаю под *подходящим местом программы?*

- Вряд ли нужно удалять окружность сразу после того, как она нарисована. Иначе у вас не будет времени, чтобы ее увидеть. Вы можете объявить объект типа Clock и задать паузу в несколько секунд (с объектами типа Clock мы познакомились на уроке 2).
- Вряд ли нужно пытаться удалить старую окружность после ее размещения на новом месте. Функция erase() фактически не удаляет объект, а закрашивает его белым цветом, поэтому, если вы размещаете окружность на новом месте, функция eraseQ будет закрашивать окружность не на старом, а на новом месте.

Кроме использования объекта типа Clock, для управления процессом удаления окружности можно рассмотреть еще две альтернативы:

- Можно удалить окружность непосредственно перед ее размещением. В этом случае при выполнении очередной итерации вы удаляете окружность перед инструкцией, которая задает ее координаты. Удаление окружности перед ее отображением на экране (перед инструкцией, которая выводит ее на экран) равносильно удалению окружности в неправильном месте.



ВНИМАНИЕ Если окружность находится в ущербном положении (0, 0), то при попытке ее стереть вы получите сообщение Object out of range (объект находится вне допустимого диапазона).

- Можно удалять окружность после выдачи запроса о необходимости продолжения рисования, но до возобновления итераций. Как это сделать? Нужно разместить запрос не в условии окончания цикла, а раньше и затем сохранить ответ. Как вы знаете, функция yesno() возвращает целое значение (1 означает *yes,* О означает *no).* Для сохранения этого ответа нужно создать дополнительную переменную:

```
int answer;
...
do
{
...
   answer = yes("Wanna try again?");
   mycircle.erase();
}
while(answer);
```

Сообщения об ошибках

Любая попытка нарисовать точку вне допустимой области экрана приведет к появлению диалогового окна с сообщением об ошибке Object out of range, которое показано на рис, 14.3. Ошибочно заданная координата (x или y) появится в левом верхнем углу экрана.

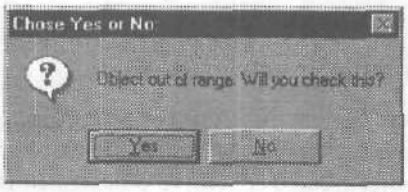


Рис. 143. Диалоговое окно с сообщением, что объект находится вне допустимого диапазона

Самостоятельная практика

- Модифицируйте программу c5circ1.cpp так, чтобы перед рисованием новой окружности программа удаляла старую.
- Модифицируйте программу c5circ1.cpp так, чтобы она рисовала не одну, а две окружности (используя два комплекта координат). Перед рисованием новых окружностей программа должна удалять старые.
- Напишите новую программу (или модифицируйте программу c5circ1.cpp) так, чтобы она рисовала несколько окружностей на горизонтальной линии. Для этого нужно запросить координаты *(x, y)* первой окружности, число окружностей и расстояние между ними. (Совет: при этом можно объявить всего одну окружность.)
- Включите в программу предыдущего упражнения объект типа Clock, чтобы рисовать окружности с двухсекундной задержкой.

Перемещение экранных объектов

Теперь, когда вы уже умеете создавать и удалять экранные объекты, мы научимся их перемещать. Как в кино или мультфильме, иллюзия движения достигается быстрой сменой изображения. Мы уже проделывали это с гимнастами на уроке 1.

Высококлассную анимацию можно получить, если выводить 30 изображений в секунду. Что означает *30 изображений в секунду*? Это означает, что за одну секунду картинка меняется 30 раз. Другими словами, каждая из них длится всего 1/30 секунды! Если картинка будет сложной, вам понадобится высокоскоростной компьютер. Однако, поскольку наши картинки достаточно простые, я не думаю, что вам придется столкнуться с этой проблемой.

Для начала разберем программу c5circ2.cpp, создающую очень простую анимацию. Если вы запустите программу, то увидите движущуюся по экрану окружность.

Попробуйте выполнить эту программу, введя 300 окружностей с координатами x=100 и y=100. Поэкспериментируйте с другими значениями.

```
#include "franca.h"                    // c5circ2.cpp
// Программа перемещения окружности по экрану:
void drawing(Cricle &mycircle, intx, int y, Box &coordx, Box &coordy)
{
   // Функция создания одного кадра изображения:
   mycircle.place(x,y);                  // Размещение окружности
   mycIrcIe.show();                      // Отображение окружности
   // Вывод значений координат в рамках:
   coordx.say(x);
   coordy.say(y);
}
void mainprog()
{
   // Объявление объектов и переменных:
   Box coordx("X:"), coordy("Y:");    // Метки информационных рамок
                                      // для ввода координат x и y
   int x, y;                          // Объявление координат
   Circle mycircle;                   // Объявление окружности
   Circle.mytimer;                    // Объявление таймера
   int howmany;
   do
   {
      // Запрос значений координат:
      x=ask("Enter x coordinate: ");
      y=ask("Enter y coordinate: ");
      // Запрос количества окружностей:
      howmany=ask("How many circles?");
      for(int k=1; k<=howmany; k++)
      {
         drawing(mycircle, x, y, coordx, coordy);
         mytimer.wait(0.033);
         mycircle.erase();
         x++;
      }
   }
   // Запрос о необходимости следующей попытки:
   while(yesno("Wanna try again?"));
}
```

Описание программы c5circ2.cpp

В программе c5circ2.cpp используются три экранных объекта:

- coordx
- coordy
- mycircle

Объекты coordx и coordy представляют собой информационные рамки, в которых отображаются значения текущих координат окружности. Объект mycircle представляет собой окружность, которая может перемещаться по экрану.

Функция drawing() создает кадр изображения. Она работает следующим образом:

- Получает в качестве параметров окружность, координаты и рамки.
- Помещает окружность в точку с координатами *(x, y)*.
- Выводит окружность на экран.
- Выводит значения координат в соответствующих информационных рамках.

Основная программа состоит из двух частей:

- Инициализация.
- Цикл.

Цикл повторяется до тех пор, пока на запрос функции yesno() пользователь отвечает *yes*. Этот цикл запрашивает ввод значений координат x и y, а также число окружностей (каждая из которых соответствует отдельному кадру изображения). Внутренний цикл вызывает функцию drawing() столько раз, сколько генерируется кадров.

Другая альтернатива состоит в том, чтобы объявить и использовать объект типа Clock не внутри цикла, а внутри функции drawing(). В этом случае функция может быть описана следующим программным кодом:

```
void drawing(Circle &mycircle, intx, inty, Box &coordx, Box &coordy)
{
   Clock mytimer;
   // Функция создания одного кадра изображения
   mycircle.place(x, y);      // Размещение окружности
   mycircle.show();           // Отображение окружности
   // Вывод значений координат в информационных рамках:
   coordx.say(x);
   coordy.say(y);
   mytimer.watch(.033);
   mycircle.erase();
}
```

Функция drawing() отображает окружность в течение 0.033 секунд, а затем стирает ее. Таким образом, функции wait() и erase() теперь можно удалить из основной функции. Как видите, может быть много вариантов создания и реализации ваших функций.



СОВЕТ Чтобы добиться хорошей анимации, минимизируйте время между удалением старого и выводом на экран нового изображения.

Самостоятельная практика

- Измените программу c5circ2.cpp так, чтобы окружность перемещалась не по горизонтали, а по вертикали.
- Измените программу c5circ2.cpp так, чтобы диаметр окружности увеличивался с каждой итерацией (начальный диаметр определите в 2 пикселя).
- Измените программу c5circ2.cpp так, чтобы она рисовала окружности разного цвета.

Смена системы координат

Все время работать в пиксельной системе координат не всегда удобно. Для смены системы координат вам может потребоваться некоторое знание геометрии.

Смена начала отсчета

Проще всего сменить начало отсчета. Если вам не нравится связывать все точки экрана с его левым верхним углом, можно переопределить начало системы координат.

Допустим, что новая точка отсчета должна располагаться на расстоянии 100 пикселей по горизонтали и 150 пикселей по вертикали от левого верхнего угла экрана. Иными словами, новые координаты начала отсчета равны $(100, 150)$. Если вы хотите поместить объект в точку с координатами (x, y) относительно новой системы координат, все, что вам нужно сделать, — это выполнить довольно простые операции с координатами (x, y) перед тем, как отображать соответствующий объект на экране.

В данном случае необходимо определить новый комплект координат $(x1, y1)$ по следующим выражениям:

```
x1 = x+100;
y1 = y+150;
```

В общем виде, если считать, что новая точка отсчета имеет координаты $(x0, y0)$, можно использовать следующие выражения:

```
x1 = x+x0;
y1 = y+y0;
```

Отметьте, когда вы хотите нарисовать объект в точке *(x, y)*, на самом деле его нужно рисовать в точке *(x+x0, y+y0)*.

Смена масштаба

Вам может потребоваться задавать расстояния не в пикселях, а в сантиметрах, дюймах или в чем-то еще. Другими словами, вы можете решить изменить масштаб.

Это тоже не сложно. Допустим, что расстояние в 100 пикселей вы хотите представить как 1000 ваших собственных новых единиц (например, вы хотите поместить 1000 футов в отрезок из 100 пикселей). Что делать? Поскольку ваш масштаб теперь 1000 единиц на 100 пикселей — или 10 единиц на пиксель, — то все, что вы должны сделать, — это разделить вашу новую координату на 10, чтобы получить координату в пикселях. Таким образом, если ваш горизонтальный масштаб будет scalex, а вертикальный — scaley, то вы можете получить новые координаты следующим образом:

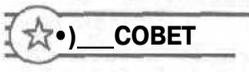
```
x2 = x/scalex;
y2 = y/scaley;
```



ПРИМЕЧАНИЕ Нельзя представить на экране объект, размер которого меньше одного пикселя. В предыдущем примере 1 пиксель выражает 10 футов. Это означает, что отрезки длиной 10, 12 или 18 футов будут выглядеть на экране одинаково.

Если вы хотите одновременно изменить масштаб и начало отсчета, то можно использовать следующие выражения:

```
x3 = (x+x0)/scalex;
y3 = (y+y0)/scaley;
```



СОВЕТ Множитель, задающий новый масштаб, получается делением числа единиц новой системы координат на эквивалентное число единиц старой.

Смена ориентации

Наконец, как сделать так, чтобы вертикальная координата была направлена не вниз, а вверх? Такое преобразование может иметь практический смысл, поскольку в большинстве математических формул предполагается именно такая ориентация осей координат.

В этом случае просто измените знак вертикальной координаты. Это может быть сделано просто умножением ее на отрицательное число. Для смены направления вертикальной оси было бы достаточно значения -1.

Рассмотрим пример создания новой системы координат, начало которой находится в нижней части экрана, а вертикальная ось направлена снизу вверх.

Мы должны сделать следующее:

- Определить координаты нового начала отсчета. Допустим, для начала отсчета мы выбираем координаты (*50, 400*).
- Определить масштаб. Для масштаба по вертикальной оси выберем отрицательное значение. Допустим, что мы хотим, чтобы единица длины составляла 60 пикселей. Тогда масштаб по соответствующим осям составит: `scalex=1/60.` и `scaley=-1/60.`.
- Переведем наши координаты в компьютерные, используя приведенные выше выражения при каждом выводе объектов на экран.

Теперь модифицировать программу `c5circ1.cpp` совсем просто. Чтобы выводить один из объектов на экран в новой системе координат, нужно изменить всего одну инструкцию:

```
mycircle.place(x, y);          // Размещение окружности
```

Новая инструкция выглядит следующим образом:

```
mycircle.place((x+50)/60., (y+400)/(-60.));
```

ВНИМАНИЕ Не забывайте при делении ставить десятичные точки! Если не показать компилятору, что в делении участвуют числа с плавающей точкой, результатом будет целое, которое в данном случае равно нулю!

Это правильное решение, однако программа, которая может разместить в определенной точке экрана только один объект, не слишком интересна. Чтобы размещать на экране различные объекты, необходимо включить новые выражения во все соответствующие инструкции.

Кроме того, если вы позже решите, что либо начало координат, либо масштаб выбраны неудачно, вам придется вводить новые значения во все необходимые инструкции. Поэтому лучшим решением было бы создание специальных функций преобразования координат. Если предположить, что следующие переменные объявлены глобально:

```
float scalex=1./60., scaley=-1/60/;
float x0=50., y0=150.;
```

тогда функции преобразования координат будут выглядеть так:

```
float newx(float oldx)
{
```

```
   return(oldx+x0)/scalex;
}
float newy(float oldy)
{
   return(oldy+y0)/scaley;
}
```

В этом случае разместить окружность на экране можно с помощью следующей инструкции:

```
mycircle.place(new(x),new(y));
```

Самостоятельная практика

- Измените программу c5circ1.cpp так, чтобы использовать новую систему координат с началом в точке (*50, 400*), масштабом 0,01 по каждой из осей и направленной вверх вертикальной осью.
- Измените программу c5circ1.cpp так, чтобы она запрашивала ввод новой точки отсчета и масштабов.

Отказ от новых функций

Оказывается, чтобы использовать новую координатную систему, вам нет нужды писать новые функции или переделывать программу. Механизм изменения системы координат уже встроен в экранные объекты.

Посылая экранному объекту сообщение scale(), можно изменить его масштаб как по горизонтали, так и по вертикали. При этом задание отрицательного масштаба для вертикальной оси будет изменять ее направление. Заметим, что в этом случае изменится масштаб всех объектов, а не только того, которому вы посылаете сообщение! Все рисунки, уже имеющиеся на экране, не изменятся, однако любой другой объект, который вы захотите разместить, удалить или отобразить, будет иметь новый масштаб.

Точно так же вы можете сменить начало системы координат, вызвав функцию-член класса экранных объектов origin(). Например, рассмотрим следующий фрагмент:

```
Circle ball;
ball.scale(1., -1.);
ball.origin(20, 400);
```

Выполнение этих инструкций приводит к тому, что начало координат перемещается в точку *(20, 400),* а вертикальная ось становится направленной вверх. Новые координаты по-прежнему выражаются в пикселях с использованием стандартного масштаба и ориентации. Если объект вызывается с новым масштабом или началом отсчета, это изменение повлияет на все последующие экранные объекты.

ВНИМАНИЕ Если вы зададите разные значения для горизонтального и вертикального масштабов, то квадраты и окружности будут деформированы! При изменении масштаба также следует размещать информационные рамки в предназначенных для них местах. Иначе механизм, автоматически создающий одну рамку под другой, может присвоить им значения координат, выходящие за границы экрана.

Объект Grid

Объект Grid может отображать горизонтальную и вертикальную координатную оси. Объявляется Grid так же, как и любой другой экранный объект. При отображении этого объекта на экране появляются горизонтальная и вертикальная координатные оси, пересекающиеся в точке начала координат *(0, ff)*.

Самостоятельная практика

- Модифицируйте программу c5circ1.cpp так, чтобы она допускала задание начала координат и масштаба, а также отображала оси координат.

Полярные координаты

В полярной системе координат положение точки определяется не координатами *x* и *y*, *а* расстоянием от точки до начала координат (которое называют *радиусом*) и углом между этим направлением и горизонтальной осью (полярным углом).

На рис. 14.4 показаны обе системы координат. Положение точки *A* может быть задано либо координатами (x, y), либо расстоянием между началом отсчета *0* и точкой *A* и углом между лучом *0A* и горизонтальной осью.

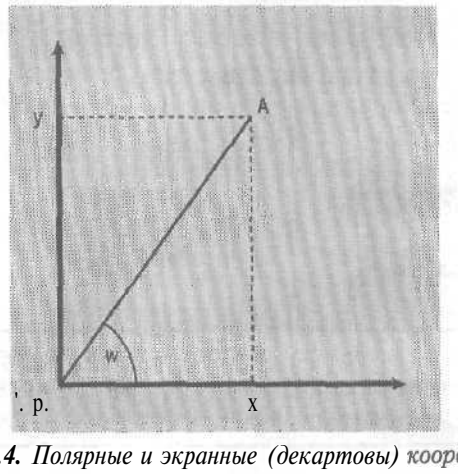


*Рис. **14.4.** Полярные и экранные (декартовы) координаты*

Полярные координаты очень полезны и удобны при описании круговых движений. С другой стороны, с помощью экранных координат легче манипулировать объектами на экране компьютера.

Рассмотрим точку *A*, определенную в полярных координатах:

Радиус: r - OA

Угол: w = XOA

Декартовы координаты этой точки можно найти по формуле:

```
x = r*cos(w);
y = r*sin(w);
```

Если точка *0* (начало отсчета полярной системы координат) в системе координат *(x, y)* имеет координаты *(x0,y0)*, то приведенные выше выражения изменяются следующим образом:

```
x = r*cos(w)+x0;
y = r*sin(w)+y0;
```

Теперь можно создать процедуру преобразования полярных координат в экранные. Если обозначить полярный угол через переменную theta, то для преобразования каждой координаты можно написать собственную функцию:

```
float coordx(float r, float theta, float x0)
{
   return r*cos(theta)+x0;
}
float coordy(float r, float theta, float y0)
{
   return r*sin(theta)+y0;
}
```

Ту же процедуру можно выполнить с помощью одной функции:

```
void polarxy(float r, float theta, float x0, float y0,
                        float &x, float &y)
{
   x= r*cos(theta)+x0;
   y= r*sin(theta)+y0;
}
```

Оба решения будут работать.



ПРИМЕЧАНИЕ В последнем случае параметры x и y передаются по ссылке (перед ними стоит символ &). Если этого не сделать, функция не будет изменять значения реальных аргументов.

Что нового мы узнали?

В этом уроке мы научились

- Задавать положение точки на экране с помощью координат.
- Работать с экранными объектами: окружностями, квадратами и информационными рамками.
- Размещать, отображать, перемещать и удалять экранные объекты в любых точках экрана.
- Изменять систему координат.

15 УРОК Создание анимаций

- Построение графиков функций
- Имитация движений на экране
- Одновременная обработка нескольких экранных объектов
- Разработка проекта планетной системы

Теперь, когда вы знаете, как, обрабатывая числовые данные, рассчитывать координаты экранных объектов, то уже можете перемещать их, создавая анимацию.

Раз вы знаете, как создавать анимацию, то можете использовать экранные объекты для построения графиков математических функций. Это очень простое приложение анимации. Здесь не нужно задумываться о скорости движения объектов. Просто в зависимости от значения горизонтальной координаты рассчитывается значение функции и откладывается по вертикальной координате. Научившись строить трафики функций, вы научитесь передавать одну функцию другой в качестве параметра.

Процесс создания анимации очень прост. Фактически он строится на повторении следующих этапов:

1. Размещение объектов в заданных точках экрана.
2. Отображение объектов в течение определенного промежутка времени.
3. Удаление объектов.

Если приходится одновременно перемещать несколько экранных объектов, то программа становится длинной и содержит слишком много повторений. Чтобы этого не произошло, мы познакомимся с новым экранным объектом — объектом типа Stage, — с помощью которого можно обрабатывать несколько объектов так, как будто это один объект.

Построение графиков математических функций

Поскольку вы знаете, как перемещать объекты по экрану, а также умеете изменять координаты и масштабы, то без труда сможете построить график функции. Мы будем строить график заданной функции y=f(x), перемещая точку (маленькую окружность) вдоль координат (x, f(x)).

Допустим, мы хотим получить график функции $y=x^2-x+1$ в интервале между точками $x=-5$ и $x=20$. Определим математическую функцию в виде функции языка C++:

```
float f(float x)
{
   return x*x-x+1;
}
```

Теперь, где бы в программе мы ни столкнулись с необходимостью вычисления функции f(x), мы всегда можем обратиться к только что созданной функции f(). Кроме того, так будет легче строить графики разных функций, о чем мы узнаем позднее.

Теперь нам осталось разместить точку в начальной позиции и затем поэтапно переместить ее в конечную, не стирая при этом промежуточные точки.

Перемещение точки

Процедура рисования функции совсем проста:

- Объявление объекта dot в качестве маленькой окружности.
- Повторение следующих шагов для различных значений x, лежащих между начальным и конечным значениями через определенные интервалы (например, 0.01).
  - Размещение объекта dot в соответствии с его новыми координатами.
  - Отображение объекта dot на экране.

На C++ это записывается следующим образом:

```
Circle dot;                    // Объявление окружности
dot.color(7, 7);               // Закрашивание окружности
dot. resize(2);                // Уменьшение окружности до точки
// Цикл от начальной позиции до конечной:
for(x=xstart; x<=xend; x=x+0.01)
{
   dot.place(x, f(x));         // Размещение точки в заданной позиции
   dot.show();                 // Отображение точки
}
```

Необходимо удостовериться, что объект на выходит за рамки экрана. При необходимости пересчитайте свои выражения или измените масштаб, чтобы были выполнены следующие условия:

- Значение x должно лежать в интервале от 0 до 640 (включительно).
- Значение y должно лежать в интервале от 0 до 480 (включительно).

Очевидно, если вы хотите пересчитывать масштабы и рисовать оси x и y, это нужно делать перед построением графика. Разумно также создать функцию из представленного выше фрагмента программы. Зачем? Функции можно использовать многократно. Если вам придется писать другую программу для построения графиков, то у вас уже будет готовая функция.

Функция draw()

Одна из возможных идей состоит в том, чтобы взять предыдущий фрагмент программного кода и сделать из него функцию. Начальное и конечное значения x войдут в нее в качестве аргументов, а систему координат будем считать заданной:

```
void draw(float xstart, float xend, int dotcolor=7)
{
   Circle dot;                          // Объявление окружности
   dot.color(dotcolor);                 // Закрашивание окружности
   dot.absize(4);                       // Уменьшение окружности до точки
   for(x=xstart; x<=xend; x=x+0.01)  // Цикл
   {
      dot.place(x, f(x));               // Размещение точки
      dot.show();                       // Отображение точки
   }
}
```

В этом случае основная программа примет следующий вид:

```
void mainprog()
{
...
   draw(-5., 20.);
}
```

Ниже представлена полная программа построения графика функции sin(x). Результат ее выполнения показан на рис. 15.1.

```
#include "franca.h"                          // c5sin.cpp
#include <math.h>
      // Программа построения графика функции sin(x)
float f(float x)
{
   return sin(x);
}
void draw(float firstx, float lastx, int color=7)
{
   float x;
   Circle dot;                               // Объявление окружности
```

продолжение

```
    dot.color(color, color);                //Закрашиваниеокружности
    dot.absize(4);                          // Уменьшение окружности до точки
    for(x=firstx; x<=lastx; x=x+0.1)        // Цикл
     {
        dot.place(x, f(x));                 // Размещение точки
        dot.show();                         // Отображение точки
     }
}
void mainprog()
{
    // Задание координатных осей, масштабов и начала координат
    Grid mygrid;
    mygrid.scale(20., -20);
    mygrid.origin(50., 300.);
    mygrid.resize(10.);
    mygrid.show();
    const float pi=3.14159;
    float xstart=0.;
    float xlimit=8.*pi;
    draw(xstart, xlimit);
}
```

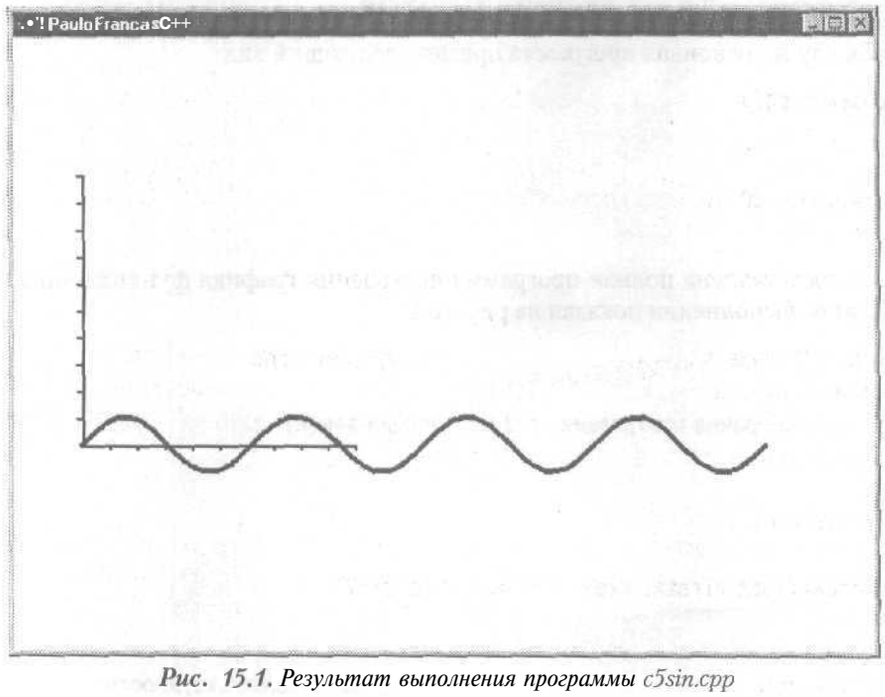


Рис. 15.1. Результат выполнения программы c5sin.cpp

Функция draw() дает достаточно разумное решение проблемы. Однако необходимо всегда задавать себе вопрос, нет ли другого, более унифицированного решения. В данном случае если вы хотите в дальнейшем еще раз воспользоваться функцией draw(), то можете столкнуться со следующими препятствиями:

- Масштаб. Можно ли включить в функцию draw() определения масштаба и осей?
- Имя функции. Что если нам понадобится строить график функции с другим именем?

Проблема с масштабом решается довольно просто. Параметры новой системы координат можно передавать функции draw() в качестве аргументов, кроме того, можно объявлять внутри функции собственный объект Grid. Но подумайте хорошенько перед тем, как так поступать. Лучше все-таки запрашивать о том, какой должна быть система координат до вызова функции.

Проблема с именем функции интереснее. Поначалу вам может показаться, что оно не имеет отношения к делу. В конце концов, можно просто изменить имя функции, сохранив ее код. Однако это неправильно. Бывает, что в одной программе необходимо строить графики сразу нескольких функций. Как же быть?

Очень просто — используйте имя функции, график которой необходимо построить, в качестве аргумента функции draw().

Функции в качестве аргументов

Язык C++ позволяет передавать функции в качестве аргументов в другие функции. Иначе говоря, если у вас имеются две функции, fl(x) и f2(x) и вы хотите построить их графики, можно переопределить функцию draw(), чтобы она получала три аргумента вместо двух. Новым аргументом будет имя функции. Тогда вызов новой функции drawf() для построения графиков двух функций fl(x) и f2(x) будет выглядеть следующим образом:

```
drawf(f1, xstart, xend);
drawf(f2, xstart, xend);
```

Эти вызовы не слишком отличаются от вызовов обычных функций. Мы задаем три аргумента: функцию (fl() или f2()), начальное значение x (xstart) и конечное значение x (xend). Осталось только написать объявление функции drawf(). Исходная функция draw() объявлялась так:

```
void draw(float xstart, float send);
```

Новая функция drawf() должна иметь еще один параметр, обозначающий функцию. А какой тип этого параметра? Мы знаем, что это не int и не float. Так каков же он? Мы только знаем, что параметром является функция.

Функции в качестве параметров

В объявление типа функции необходимо включить следующие пункты:

- Тип возвращаемого значения (например, float, void и т. п.).
- Символическое имя функции (например, func).
- В круглых скобках список типов аргументов (например, (float)).

В нашем примере описание функции drawf() может быть следующим:

```
void drawf(float func(float), float firstx, float lastx);
```

Обратите внимание на три параметра функции:

- Функция, получающая переменную типа float в качестве аргумента и возвращающая тоже значение типа float. Эта функция обозначена символическим именем func.
- Две переменные с плавающей точкой. Эти переменные обозначены символическими именами firstx и firsty.

Полный код функции приведен ниже:

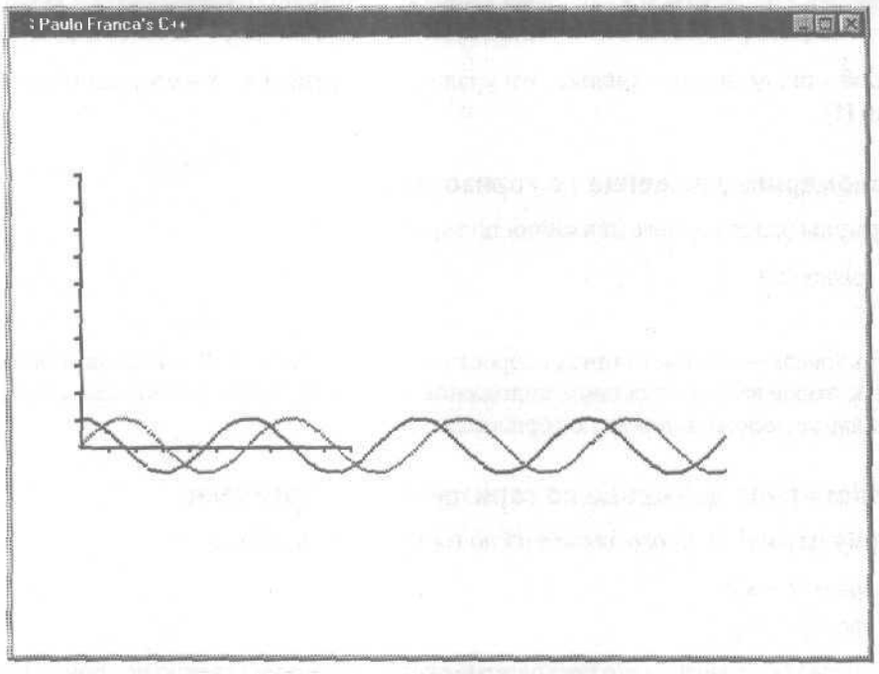
```
void drawf(float func (float), float firstx, float lastx)
{
   float x;
   CircIe dot;                          // Объявление окружности
   dot.color(7, 7);                     // Закрашивание окружности
   dot.resize(2);                       // Уменьшение окружности до точки
   for(x=fistx; x<=lastx; x=x+0.01)     // Цикл
   {
      dot.place(x, func(x));            // Размещение точки
      dot.show();                       // Отображение точки
   }
}
```

Когда мы указываем местоположение точки, используя аргументы x и func(x), то функция func() будет вызвана с аргументом x. Затем результат вызова функции становится вертикальной координатой точки.

Самостоятельная практика

- Разработайте программу построения графиков функций sin(x) и cos(x). График функции sin(x) должен быть нарисован красным цветом, а функции cos(x) —

синим. Совет: доступ к функциям sin(x) и cos(x) достигается включением в программу заголовочного файла math.h. Результат выполнения программы должен выглядеть так, как показано на рис. 15.2.



Рис, 15.2. *Графики функций синуса и косинуса*

Создание анимации

Теперь воспользуемся экранными объектами для создания анимации. С простейшей анимацией мы уже сталкивались в программе c5circ.cpp. В этом разделе с помощью экранных объектов мы проиллюстрируем некоторые уравнения движения, известные нам из курса физики,

Иллюзия движения

Имитировать движение на экране компьютера можно тем же путем, которым делаются фильмы, — отображением чередующихся образов через небольшие промежутки времени. Такой подход позволяет создавать разные виды анимации.

Один из простых способов создать анимацию состоит в том, чтобы выразить все движения математическими формулами. Эти формулы можно встретить во многих физических задачах, некоторые из них описываются уравнениями, задающими положение тела.

Имитация сложных движений

Допустим, у вас имеется движущееся тело, зависимость координат x и y которого от времени определяется законами x=dist(y) и y=height(t). Координата x задает горизонтальное расстояние от начала отсчета, а координата y — высоту над землей.

Как обычно делается в физике, эти уравнения определяют x и y как функции времени (t).

Равномерное движение по горизонтали

Формулы равномерного движения по горизонтали:

```
x = speedx*t + xO;
y = yO;
```

Здесь speedx — это постоянная скорость по горизонтали, а yO — постоянное значение, которое задает начальное положение тела по вертикали. Константа xO задает начальное положение тела по горизонтали.

Равномерное движение по горизонтали и вертикали

Формулы равномерного движения по горизонтали и вертикали:

```
x = speex*t + xO;
y = speedy*t + yO;
```

Здесь speedx и speedy — это постоянные скорости соответственно по горизонтали и вертикали.

Равноускоренное движение

Если тело движется с постоянным ускорением, то уравнение описывается следующими строками кода:

```
x = accx*t*t/2 + speedx*t + xO;
y = accy*t*t/2 + speedy*t + yO;
```

Если вы посмотрите внимательно, то поймете, что после создания функций для определения расстояния по горизонтали dist(t) и высоты height(t) имитация движения уже не будет зависеть от того, какое движение вы пытаетесь имитировать.

Если предположить, что эти функции будут написаны правильно, то разработать программу создания анимации можно с помощью следующей стратегии:

- Объявить экранный объект, который будет представлять движущееся тело (например, окружность).
- Объявить объект типа Clock, чтобы была возможность отслеживать время.
- Создать соответствующий цикл, чтобы между появлением двух соседних кадров проходил определенный промежуток времени (для качественной анимации это должна быть 1/30 секунды). Цикл должен выполнять следующие действия:

1. Вычислять новые координаты как функции *времени*.
2. Перемещать тело в новое положение.
3. Отображать его на экране.
4. Ждать определенный временной промежуток перед отображением следующего кадра (1/30 секунды),
5. Удалять тело с экрана.

Все это будет выполнять наша следующая программа.

Свободное падение

В программе c5body.cpp для вычисления координат x и y движущегося тела имеются функции dist(t) и height(t). Текущие координаты движущегося тела и текущее значение времени выводятся на экран в соответствующих информационных рамках.

```
#include "franca.h"                    // c5body.cpp
#include "math.h"
Grid migrid;
//' Использование стандартных координат
const float v0x=10., v0y=20.;
float dist (float t)                   // Функция определения координаты x
{
   return v0x*t+10;
}
float height (float t)                 // Функция определения координаты y
{
   return v0y+50;
}
void mainprog()
{
   // Объявление и именование информационных рамок
   // для отображения координат x, y и времени
   Box boxx("x:"), boxy("y:"), boxt("Time:");
   mygrid.show();                      // Рисование координатных осей
   float t;                            // Объявление переменной времени
   Circle body;                        // Объявление тела
   body. resize(12);                   // Задание размера тела
   body.color(3,3);                    // Закрашивание тела
   Clock timer;                        // Объявление таймера
   Clock mywatch;                      // Объявление другого таймера
   // Цикл будет выполняться до тех пор, пока не пройдет 15 секунд;
```

```
    // текущее время накапливается в переменной t
    while((t=mywatch.time())<15.)
    {
       // Обновление значений координат и времени
       // в соответствующих информационных рамках:
       boxx.say(dist(t));
       boxy.say(height(t));
       boxt.say(t);
       // Размещение тела в текущем положении:
       body.place(dist(t), height(t));
       body.show();                  // Отображение тела
       timer.watch(.033);            // Пауза в 0.033 секунды
       timer.reset();                // Перезапуск таймера
       body.erase();                 // Удаление тела с экрана
    }
}
```

Эта программа воплощает описанную выше стратегию, и вы уже должны понимать, как она работает. Тем не менее, возможно, будет полезно специально объяснить некоторые аспекты.

Два объекта типа Clock и функция-член watch ()

В программе объявляются два объекта типа Clock — это mywatch и timer. Один мы используем для постоянного отсчета времени (mywatch), а другой — в качестве секундомера (timer). Именно объект типа timer позволяет каждые 0.033 секунды отображать новый кадр.

Функция-член watch() больше подходит для имитации движения, чем функция-член wait(), поскольку может задавать более точные интервалы времени. При задании интервала времени с помощью функции wait() к этому интервалу будет неявно прибавляться время, необходимое компьютеру для подготовки очередного кадра. Функция watch() свободна от этого недостатка.

Выражение в инструкции while

В инструкции while мы разместили необычное выражение. Оно позволяет, во-первых, продолжать итерации, пока значение переменной time меньше 15, во-вторых, копирует значение переменной time в переменную t. Эту задачу можно было бы решить в два этапа:

```
while(mywatch.time()<15)
{
   t = mywatch.time();
...
```

Такая запись формально не содержит ошибки, однако, как вы могли заметить, значение, присваиваемое переменной t, немного больше значения, тестируемого в выражении инструкции while (поскольку на выполнение этой инструкции тоже уходит какое-то, пусть и небольшое, время).

Выражение, которое использовалось в программе, вы могли бы попытаться заменить следующим;

```
while(t=mywatch.time()<15)
```

К несчастью, это выражение тоже ведет к неправильному результату, поскольку в этом случае изменяется порядок выполнения операций. Сначала значение переменной time сравнивается с 15. Результатом будет 1, что означает *истина*. Затем этот результат присваивается переменной t Таким образом, пока значение переменной time будет оставаться меньше 15, значение переменной t будет равно 1.

Обработка более фундаментальных движений

В предыдущем примере мы работали с функциями, задающими уравнения равномерного движения по обеим координатам. В результате при запуске программы мы видели окружность, движущуюся по прямой линии с постоянной скоростью. Однако та же программа может использоваться для задания движения любого вида. Все, что нам нужно сделать, — это изменить функции, определяющие закон движения!

В качестве примера давайте разберем движение падающего тела, брошенного параллельно поверхности земли. При этом движение по горизонтали по-прежнему происходит с постоянной скоростью и описывается предыдущей формулой. По вертикали же уравнение для координаты y примет следующий вид:

$$y = h_0 + V_{oy}t - \frac{1}{2}gt^2$$

или на C++

```
y = h0+v0y*t+0.5*g*t*t;
```

При этом предполагается, что:

- h0 — начальная высота при t=0.
- vOy — начальная скорость по вертикали при t=0.
- g — ускорение свободного падения (9,8 м/с^2).
- Тело бросается с нулевой вертикальной скоростью, v0y=0.

Функция height(t) может быть определена следующим образом:

```
float height(float t)
{
   const float h0=200., g=32.18;
   return h0+0.5*g*t*t;
}
```

ПРИМЕЧАНИЕ В функциях, о которых рассказывается в этом разделе, кроме времени могут быть задействованы и другие переменные (например, h0, vOy и g). Если это так, старайтесь использовать эти переменные в качестве аргументов функции. В предыдущем примере vOx и vOy мы определили как глобальные константы. Но если практика объявления *глобальных констант* не может повредить вашим программистским навыкам, то вот практики объявления *глобальных переменных* следует всячески избегать.

В программе c5body.cpp замените функцию height(t), запустите программу и посмотрите результат. Но если вы сделаете только это, тело будет падать до самых границ экрана. Ведь никакого «пола», чтобы остановить его, нет. Поэтому разумно было бы контролировать высоту тела и останавливать программу, когда она становилась бы отрицательной. Здесь имеются две альтернативы:

- Вы можете расширить условие в инструкции while, чтобы цикл повторялся до тех пор, пока не истекло **15** секунд *и* пока высота больше или равна нулю. Например:

```
while(((t=mywatch.time(t))<15.)&&(height(t)>=0))
```

ВНИМАНИЕ Не забывайте про круглые скобки!

- Можно контролировать высоту внутри цикла и выходить из него в случае отрицательного результата. Например, можно добавить в тело цикла следующую инструкцию:

```
if (height(t)<=0) break;
```

Самостоятельная практика

- Модифицируйте программу c5body.cpp так, чтобы имитировать свободное падение тела. Останавливайте программу, когда высота тела становится равной нулю.

Вопросы эффективности

Как мы знаем, компьютеры работают очень быстро. Однако и сам программист должен стараться избавить компьютер от ненужной работы, чтобы все необходимые вычисления могли укладываться в положенное время. Если вернуться к программе c5body.cpp, можно заметить, что функции dist(t) и height(t) при каждой итерации вызываются дважды. В этом нет ничего плохого за исключением того, что в обоих случаях переменная t имеет одно и то же значение — оба вызова выполняются с одним и тем же значением!

Разумно ли так нагружать компьютер? Конечно нет. Если вы вычислили какое-либо значение и собираетесь использовать его в программе несколько раз, лучше сохранить это значение в переменной, а не снова и снова его рассчитывать. В нашем случае сначала необходимо объявить две дополнительные переменные:

```
float x, y;
```

Затем следующим образом модифицировать цикл:

```
while(t=mywatch.time())<15.)
{
   x=dist(t);
   y=height(t);
   // Обновление значений координат и времени
   // в соответствующих информационных рамках:
   boxx.say(x);
   boxy.say(y);
   boxt.say(t);
   body.place(x, y);      // Размещение тела
}
```

А если вы хотите сделать нечто действительно необычное, можете присваивать значения прямо при отображении значений в информационных рамках:

```
boxx.say(x=dist(t));
boxy.say(y=height(t));
```



ПРИМЕЧАНИЕ Этот вариант не дает существенной экономии компьютерных ресурсов, но может сделать вашу программу менее понятной.

Еще один аспект эффективности — выражения с константами (например, 1./2). Каждый раз, встречая это выражение в программе, компьютер будет заниматься делением. Можно сэкономить ресурсы, используя эквивалентное значение 0.5. Можно также объявить константу или переменную требуемого значения. Например:

```
const float pi2=3.14159/2
```

Теперь, вместо деления константы pi на число 2, компьютер будет использовать в программе константу pi2.

Имитация пушечного ядра

Представим себе пушку, стреляющую ядрами под разными углами. Вектор скорости пушечного ядра в момент вылета из ствола направлен в направлении стрельбы.

Это направление задается углом theta. Как вы знаете из курса физики, вектор скорости можно разложить на проекции:

```
velocx = veloc*cos(theta);
velocy = veloc*sin(theta);
```

Движение ядра может быть, таким образом, представлено через знакомые нам функции dist(t) и height(t). При этом движение по горизонтали происходит с постоянной скоростью (равномерное движение), в то время как по вертикали движение определяется ускорением свободного падения.

Функции могут быть определены следующим образом:

```
const float g=32.18;          // Ускорение свободного падения
// Функции вычисления координат
float dist(float velocx, float t)
{
   return velocx*t;
}
float height(float velocy, float t)
;
   return velocy*t - 0.5 * g * t * t;
}
```

Вы можете определить g как глобальную константу.

Основная программа будет запрашивать ввод угла theta и начальной скорости veloc. Затем программа, вычислив значения velocx и velocy, уже сможет имитировать движение, как было показано в предыдущих программах.



ПРИМЕЧАНИЕ Не забывайте, что тригонометрические функции требуют задания углов в радианах, поэтому необходимо выполнить соответствующее преобразование.

Вычисление координат ядра

Программа c5cannon.cpp моделирует движение ядра. Программу образуют следующие основные разделы:

- Функции определения расстояния и высоты.
- Задание начальных условий.
- Моделирующий цикл.

Функции мы уже рассматривали. В установку начальных условий входит задание масштабов, отображение осей x и y, создание окружности, изображающей ядро, и

инициализация часов, необходимых для имитации. Только один из этих пунктов может вас смутить — задание масштабов.

Масштабы должны быть выбраны так, чтобы все объекты, которые мы хотим изобразить, поместились на экране. Необходимо проверить функции height() и dist() для определения максимальных значений x и y. К несчастью, эти значения зависят от значений скорости и угла, которые вводит пользователь. Один из аспектов, значительно затрудняющих программирование, состоит в том, что вы не всегда знаете, кто и как будет использовать ваши программы!

Лучшее, что мы можем, — это сделать разумную оценку. Если мы допустим скорость ядра в 1000 футов в секунду, то можем сказать, что дальность его полета не превысит 30 000 футов при начальном угле в 45° (что дает максимальную дальность полета). Если же пушка стреляет под углом 90° (странное желание), то максимальная высота ядра будет 15 000 футов.

Поскольку по обеим осям откладывается расстояние, разумно использовать один и тот же масштаб по вертикали и горизонтали. Таким образом, мы можем принять 30 000 футов за максимальные значения расстояния по горизонтали и высоты.

Расчет для VGA

С другой стороны, несмотря на то что экран обычно содержит 640x480 пикселей, разумно несколько ограничить изображение, сделав его, например, 400x400 пикселей. Это означает, что наши 30 000 футов должны уместиться на 400 точках экрана. В результате наш масштаб будет равен 400/30 000 точек на фут.



СОВЕТ

Если при расчете масштабов вы не уверены в своих способностях, вам может помочь знание размерностей. В предыдущем случае 30 000 футов приходилось на 400 точек. Сначала вам может быть непонятно, что на что нужно делить: 400/30 000 или 30 000/400? Если же вы посмотрите на размерность, которая в первом случае выражается в точках на футы, в во втором — в футах на точки, то заметите, что при умножении расстояния (в футах) на масштаб во втором варианте вместо точек мы получим квадратные футы на точку.

Моделирующий цикл

Наиболее интересная часть программы — моделирующий цикл. Итерации повторяются до тех пор, пока не истекает 50 секунд. В каждом цикле новые координаты вычисляются для нового значения времени. Затем старое ядро удаляется и рисуется уже на новом месте. Потом задается пауза на 0.033 секунды, чтобы мы могли наблюдать картинку на экране. После окончания этого интервала секундомер устанавливается на ноль и цикл повторяется.

Ниже приведена основная функция программы c5cannon.cpp.

```
void mainprog()                    // Часть 1
{
   Grid mygrid;
   mygrid.origin(50.,420);
   mygrid.scale(400/30000, -400/30000);
   // Объявление и именование информационных рамок
   // для отображения координат x, y и времени
   Box boxx("x:"), boxy("y:"), boxt("Time:"), Box vx("Vx:"), vy("Vy:");
   mygrid. resize(30000.);
   mygrid. show();                 // Рисование координатных осей
   float t;                        // Объявление переменной времени
   Circle body;                    // Объявление тела
   body.absize(10);                // Задание размера тела
   body.color(3,3);                // Закрашивание тела
   Clock timer;                    // Объявление таймера
   Clock mywatch;                  // Объявление другого таймера
   float theta, veloc, velx, vely;
   // Запрос значения угла
   theta=ask("Enter angle of firing:");
   theta= theta*3.14159/180.;      // Преобразование угла в радианы
                                   // Запрос значения начальной скорости
   veloc=ask("Enter initial velocity:");
   velx=veloc*cos(theta);          // Преобразование в координаты x и y
   vely=veloc*sin(theta);
   vx.say(velx);
   vy.say(vely);
   float x, y;                     // Часть 2
   // Цикл будет выполняться до тех пор, пока не пройдет 15 секунд;
   // текущее время накапливается в переменной t
   timer. reset();
   watch.reset();
   while((t=watch.time())<50.)
   {
      y=height(vely, t);
      if(y<0) break;
      x=dist(velx, t);
      // Обновление значений координат и времени
      // в соответствующих информационных рамках;
      boxx.say(x);
      boxy.say(y);
      boxt.say(t);
      body.erase();
      body.place(x, y);            // Размещение тела
      body.show();                 // Отображение тела
      timer.watch(.033);           // Пауза в 0.033 секунды
      timer. reset 0;              // Перезапуск таймера
   }
}
```

На рис. 15.3 — 15.5 показаны результаты выполнения программы, моделирующий полет ядра через разные промежутки времени.

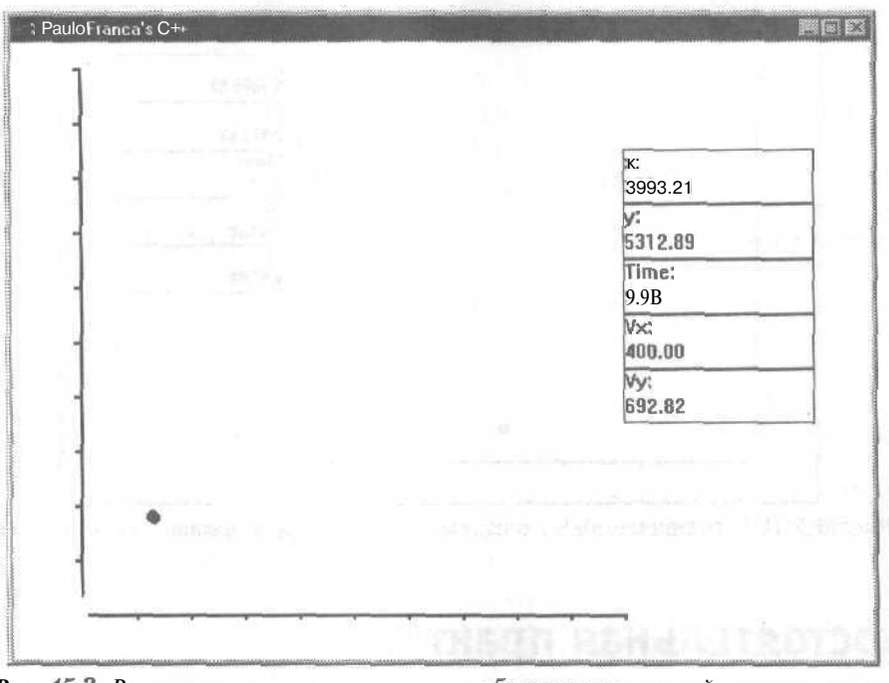


Рис. 15.3. *Результат выполнения программы c5cannon.cpp в первый момент времени*

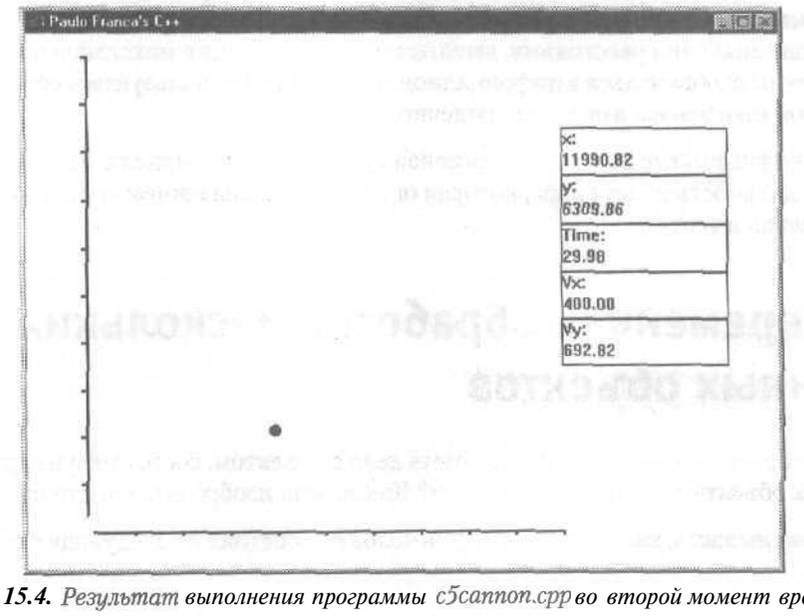


Рис. 15.4. *Результат выполнения программы c5cannon.cpp во второй момент времени*

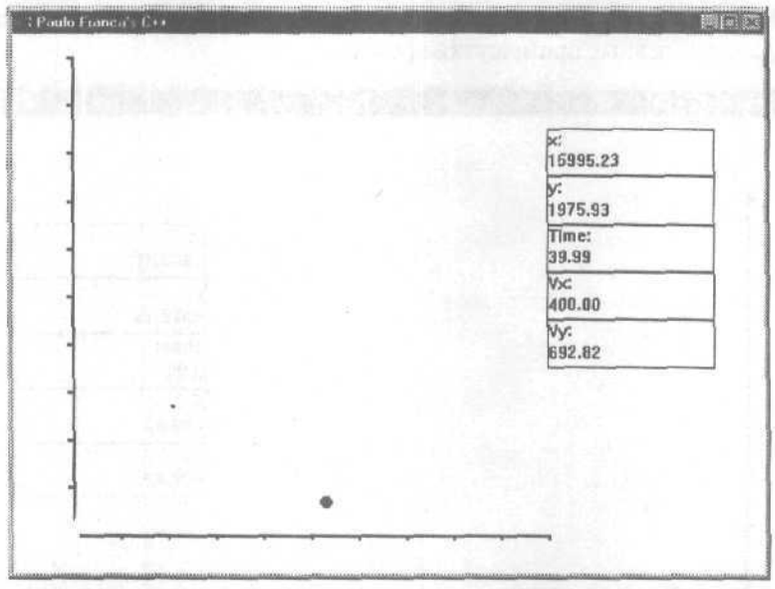


Рис. 15.5. *Результат выполнения программы c5cannon.cpp в третий момент времени*

Самостоятельная практика

- Используя функции dist(t) и height(t) составьте программу, которая находила бы максимальное значение высоты под данным значением скорости и угла. Текущие значения расстояния, высоты и времени, а также максимальной высоты должны отображаться в информационных рамках. Воспользуйтесь общей структурой программы имитации пушечного ядра.
- Модифицируйте программу c5cannon.cpp так, чтобы вычислялась максимальная дальность полета ядра, которая определяется значением координаты x при нулевой высоте.

Одновременная обработка нескольких экранных объектов

В некоторых случаях приходится иметь дело с объектом, состоящим из других экранных объектов. Помните гимнастов? Как можно изобразить одного из них?

Фигура гимнаста, как и всякая фигура человека, состоит из следующих частей:

- Головы (окружность).
- Тела (квадрат).
- Левая и правая рука (прямоугольники).
- Левая и правая нога (прямоугольники).

Для простоты размер тела будем определять только через два значения. Назовем их L и W, как показано на рис. 15.6. Голова представляет собой окружность радиусом W. Руки и ноги представлены прямоугольниками шириной W и длиной L. Туловищем является квадрат со стороной L.

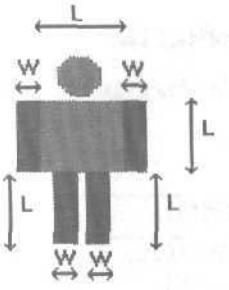


Рис. 15.6. *Изображение гимнаста*

Построение гимнаста

Можете ли вы нарисовать гимнаста? Есть два пути: трудный и легкий. Вы можете начать с объявления объектов, образующих части тела (трудный путь!). Позднее вы научитесь использовать класс Stage (легкий путь!).

```
Circle head;
Square trunk;
Square leftleg, rightleg;
Square leftarm, rightarm;
```

Поскольку теперь мы умеем обращаться с окружностями и квадратами, то нам остается лишь поместить эти объекты в нужное положение и присвоить им соответствующие цвета и размеры.

Допустим, что гимнаст находится внутри квадрата, центр которого имеет координаты x и y. Кроме того, для значений W и L давайте объявим константы, соответственно armwidth и armsize.

```
const int armsize = 20;
const int armwidth = 6;
```

Закрашивание гимнаста

Для закрашивания объектов воспользуемся следующими инструкциями:

```
head.color(5, 5);
trunk.color(5, 5);
leftarm.color(3, 3);
rightarm.color(3, 3);
leftleg.color(3, 3);
right.color(3, 3);
```

Масштабирование гимнаста

Теперь можно присвоить всем объектам соответствующие размеры:

```
head, resize(2*armwidth);
trunk. resize(armsize);
leftarm. resize(armsize, armwidth);
rightarm. resize(armsize, armwidth);
leftleg. resize(armsize, armwidth);
rightleg. resize(armsize, armwidth);
```

Размещение гимнаста

Разместим все объекты:

```
head.place(x, y-armsize/2.-armwidth);
trunk.place(x ,y);
leftarm.place((x-(armsize+armwidth)/2.), y);
rightarm.place((x+(armsize+armwidth)/2.), y);
leftleg.place((x-(armsize/2.-armwidth), y+armsize);
rightleg.place((x+(armsize/2.-armwidth), y+armsize);
```

Отображение гимнаста

Теперь наконец все объекты можно вывести на экран:

```
head.show();
trunk.show();
leftarm.show();
```

```
rightarm.show();
leftleg.show();
rightlet.show();
```

Попробуйте нарисовать гимнаста с помощью этих инструкций. Однако лучше наберитесь терпения и в следующем разделе изучите класс Stage.

Класс Stage

Класс Stage предназначен для обработки группы связанных экранных объектов.

Объекты класса Stage принадлежат еще и классу ScreenObj (класс экранных объектов). Следовательно они, как и любые экранные объекты, имеют координаты, их можно отображать на экране и удалять с экрана. Однако экранные объекты типа Stage не имеют формы. Но вы можете вставить несколько экранных объектов в объект типа Stage, а затем с помощью функции show() отобразить на экране все объекты, входящие в объект типа Stage.

В дополнение к обычным операциям над экранными объектами, над объектом типа Stage можно выполнить операцию insert (вставка), с помощью которой любые экранные объекты можно вставить в объект Stage.



ПРИМЕЧАНИЕ Можно также вставить объект типа Stage внутрь другого объекта типа Stage.

Как мы уже знаем, гимнаст состоит из нескольких объектов. Если вы хотите перемещать гимнаста по экрану, то должны выполнить следующие операции над каждым из составляющих его объектов:

- Удалить прежнее изображение объекта.
- Переместить объект в новое положение.
- Отобразить объект в новом положении.

Объекты класса Stage

Использование класса Stage позволяет выполнять некоторые операции над всеми объектами, входящими в объект типа Stage, посылая соответствующие сообщения только объекту типа Stage! Иными словами, вместо того чтобы задавать положение головы, левой руки, правой ноги и т. д., можно задать положение всего гимнаста:

```
Stage body;
body.place(x, y);
```

Важно правильно разместить все объекты внутри объекта типа Stage, поскольку иначе сам объект типа Stage окажется кривобоким!

Все, что нам нужно, — это включить каждый объект внутрь объекта типа Stage (только один раз):

```
body.insert(trunk);
body.insert(leftarm);
body.insert(rightarm);
body.insert(leftleg);
body.insert(rightleg);
```

Если мы захотим переместить объект типа Stage в новое положение x1, y1, то это можно сделать следующим образом:

```
body.erase();
body.place(x, y);
body.show();
```

Конечно же это гораздо проще, чем удалять, размещать и отображать каждую деталь в отдельности.

Проект планетной системы

В заключение урока 15 попробуем смоделировать простейшую планетную систему. Допустим, ваш преподаватель по физике в качестве домашнего задания попросил вас разработать наглядную анимацию планетной системы, в которой Земля вращается вокруг Солнца, а Луна вращается вокруг Земли.

Нет никакого смысла сохранять правильные пропорции между размерами объектов, поскольку в этом случае как Земля, так и Луна на экране окажутся невидимыми!

Вместо этого для каждого тела можно выбрать произвольный размер (диаметр), равно как оставить произвольными расстояния между телами. Мы также будем имитировать круговые движения, хотя на самом деле они эллиптические.

Тем не менее важно сохранить временные пропорции. Например, одна секунда при моделировании может реально соответствовать одному дню. Однако для наших целей это, вероятно, слишком медленно — весь цикл моделирования, включающий 365 дней, будет длиться 365 секунд. Студенты, которым вы соберетесь показать все это, должны будут терпеливо сидеть и наблюдать за этим круговоротом светил! Поэтому лучше считать, что реальный день соответствует одной десятой секунды.



ПРИМЕЧАНИЕ Создавая этот проект, помните, что Земля совершает полный оборот вокруг Солнца за 365.25 дней, а Луна полный оборот вокруг Земли — за 28 дней.

После объявления и инициализации Солнца (объект Sun), Земли (объект Earth) и Луны (объект Moon) в качестве окружностей (объектов класса Circle) выполнить само моделирование уже не сложно.

Положение каждого тела зависит только от времени. Каждое тело должно быть удалено со своей прежней позиции, размещено в новой и затем вновь отображено, Использование полярных координат в этом случае может принести реальную

пользу. Поскольку угловые скорости Земли и Луны известны, то угол может быть вычислен как функция времени.

Объекты для проекта планетной системы

Представить Солнце, Землю и Луну через объекты класса Circle не трудно. Их нужно просто объявить следующим образом:

```
Circle Sun, Earth, Moon;
```

Далее эти объекты должны быть инициализированы соответствующими значениями цвета, размера и начального положения.

Удобное начальное положение показано на рис. 15.7. В этом случае в начале моделирования тела находятся на горизонтальной линии. Какуже упоминалось, все расстояния между ними произвольны, и было бы неплохой идеей использовать именованные константы, чтобы подобрать подходящие значения расстояний и размеров.

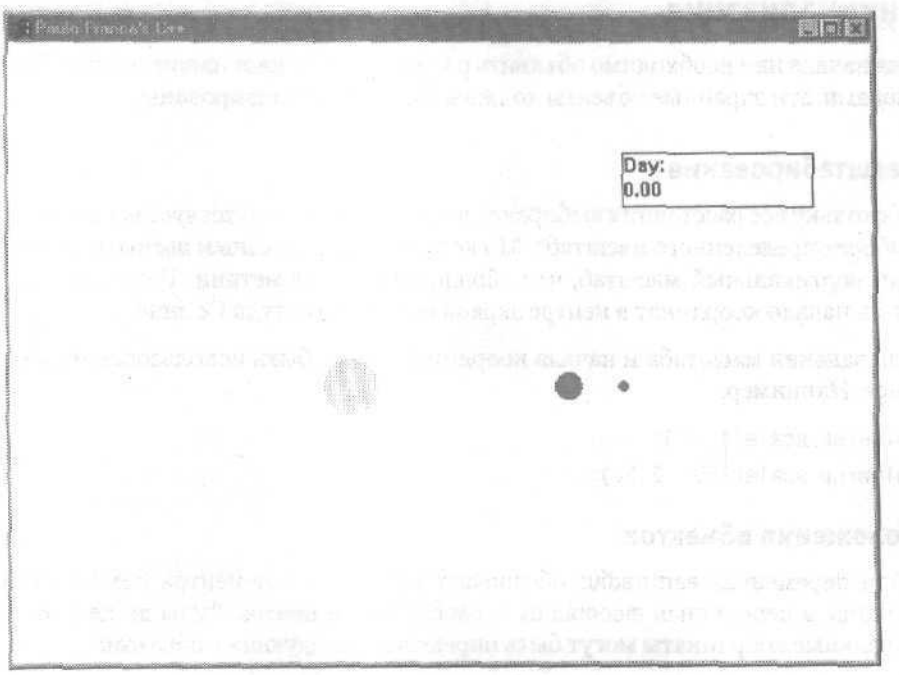


Рис. 15.7. Начальное положение Солнца, Земли и Луны

Для начала можно попытаться использовать следующие значения:

```
const float earthradius=160., moonradius=40.;
const float earthsize= 20., moonsize=8., sunsize= 40.;
```

Не забудьте о классе Stage

Хотя поначалу это не кажется очевидным, можно попытаться в качестве всех планет использовать один объект типа Stage. В процессе моделирования вы будете

удалять и вновь отображать эти объекты. Если их включить в объект типа Stage, то это сэкономит вам несколько строк программного кода. Я предлагаю следующие инструкции:

```
Stage universe;
universe. insert(Sun);
universe. insert(Earth);
universe. insert(Moon);
```

Вам также понадобится объект типа Clock, чтобы иметь возможность контролировать время. На самом деле лучше объявить два таких объекта. Значение одного из них, контролирующего время в общем процессе моделирования, можно выводить на экран, чтобы пользователь видел текущий день года. С помощью другого объекта типа Clock можно сменять кадры через интервалы в 0.033 секунды.

```
Clock universal, clock stopwatch;
```

Инициализация

Для начала нам необходимо объявить размер, место и цвет наших планет. Другими словами, эти экранные объекты должны быть инициализированы.

Масштабирование

Поскольку все расстояния выбираются произвольно, отсутствует необходимость в выборе определенного масштаба. Может оказаться полезным выбрать отрицательный вертикальный масштаб, что общепринято в геометрии. Разумно также выбрать начало координат в центре экрана и поместить туда Солнце.

Для задания масштаба и начала координат может быть использован объект типа Stage. Например:

```
universe.scale(1, -1);
universe.scale(250., 250.);
```

Положения объектов

Если переменная earthradius обозначает расстояние от центра Земли до центра Солнца, а переменная moonradius — расстояние о центра Луны до центра Земли, начальные координаты могут быть определены следующим образом:

Солнце: x=0; y=0;

Земля: x=earthradius; y=0;

Луна: x= earthradius+moonradius; y=0;

Полностью инициализация описана ниже:

```
const float earthradius=160., moonradius=40;
const float earthsize=20., moonsize=8., sunsize=40.;
Stage universe;
universe.origin(250., 250.);
```

```
universe.scale(1., -1.);
// Установка данных для Солнца;
Circlesun;                                  // Объявление формы Солнца
float xsun=0, ysun=0;                       // Начальные координаты
sun.resize(sunsize);
sun.color(6, 6);
sun.place(xsun, ysun);
universe.insert(sun);
// Установка данных для Земли
Circle earth;                               // Объявление формы Земли
float xearth= earthradius, yearth=0;        // Начальные координаты
float wearth=(2*pi)/365.25;                 // Угловая скорость
                                            // в радианах за день
earth.resize(earthsize);
earth.color(5, 5);
earth.place(xsun+ earthradius, ysun);
universe.insert(earth);
// Установка данных для Луны
Circle moon;
float xmoon= moonradius, ymoon=0;
float wmoon=(2*pi)/27;                      // Угловая скорость
moon.resize(moonsize);
moon.color(1, 1);
moon.place(xsun+ earthradius+moonradius, ysun);
universe.insert(moon);
```

Моделирующий цикл

Как обычно, моделирующий цикл представляет собой генерирование кадров через определенные интервалы времени. Перед отображением Земли и Луны на экране их необходимо переместить в новые положения.

Общая процедура моделирующего цикла:

1. Удалить Вселенную (объект universe типа Stage).
2. Вычислить новое положение Земли.
3. Разместить Землю в новом положении.
4. Вычислить новое положение Луны.
5. Разместить Луну в новом положении.
6. Отобразить Вселенную.
7. Вывести текущее время.
8. Сделать паузу и повторить цикл.

Положение Земли

Можете ли вы найти положение Земли в заданный момент времени? Не забывайте, что Земле требуется 365.25 дней, чтобы совершить полный оборот вокруг Солнца.

Вы можете рассчитать угловую скорость или в градусах, или в радианах:

w = 365.25/360 градусов в день;

w = 365.25/(2*pi) радиан в день.

С помощью угловой скорости Земли (переменная wealth) можно найти угловое положение Земли (переменная thetaearth) в любой день (переменная time):

```
thetaearth = wearth*time;
```

Поскольку Солнце находится в начале координат, можно использовать расстояние от центра Земли до центра Солнца (константа earthradius) и найденное угловое положение Земли в качестве полярных координат для вычисления декартовых координат Земли.

Чтобы получить значения декартовых координат Земли xearth и yearth, теперь достаточно вызвать функцию polarxy:

```
polarxy(earthradius, wearth*time, xsun, ysun, xearth, yearth);
```

Положение Луны

Аналогичным образом можно определить декартовы координаты Луны. Помните, что требуется всего-навсего определить угол. Вызывая функцию polarxy, рассматривайте в качестве начала координат Землю:

```
polarxy(moonradius, wmoon*time, xearth, yearth, xmoon, ymoon);
```

После определения положений Земли и Луны можно наконец представить моделирующий цикл нашей планетной системы в виде программного кода:

```
for(;time<365.25;)
{
   universe.erase();
   polarxy(earthradius, wearth*time, xsun, ysun, xearth, yearth);
   eath.place(xearth, yearth);
   polarxy(moonradius, wmoon*time, xearth, yearth, xmoon, ymoon);
   moon.place(xmoon, ymoon);
   day.say(time);
   universe.show();
   stopwatch.watch(.033);
   stopwatch.reset();
   time=sideral.time()*timescale;
}
```

В этом цикле имеется переменная timescale, определяющая скорость движения объектов. Равное 10 значение переменной timescale означает, что одна секунда соответствует десяти дням. В данном цикле проверяется только одно условие, не прошло ли более 365.25 дней (не завершился ли год).

Полную версию программы можно найти на прилагаемой дискете в файле c5stars.cpp.

Самостоятельная практика

- Модифицируйте программу c5stars.cpp так, чтобы на экране была видна орбита Луны. Для этого просто исключите в цикле удаление Луны.
- Модифицируйте программу c5stars.cpp так, чтобы фон имел голубой цвет вместо белого.

Что нового мы узнали?

В этом уроке мы научились

- Перемещая объекты, строить графики функций.
- Перемещая объекты через определенные промежутки времени, имитировать на экране движение реальных объектов.
- Использовать класс Stage, чтобы одновременно обрабатывать несколько экранных объектов.
- Строить модель планетной системы.

Часть VI

Классы

Теперь вы уже умеете использовать объекты, но только готовые, поскольку создавать собственные классы объектов вы еще не научились. В части VI вы узнаете, как создавать классы объектов и даже как строить их из уже существующих классов, *наследуя* при этом все интересующие вас свойства. В этом вам снова будут помогать экранные объекты. Одновременно мы будем продолжать совершенствоваться в создании приложений.

16 УРОК Создание и модификация классов

- Объявление и определение класса
- Открытые, закрытые и защищенные члены класса
- Защита информации с помощью инкапсуляции
- Конструкторы
- Использование объектов
- Создание нового класса объектов
- Доступ к членам класса

С объектами и классами мы знакомились, начиная с урока 1. Сейчас самое время узнать о них побольше.

Вы уже знаете, что каждый класс объектов может реагировать на строго определенные сообщения. Например, для объектов класса athlete это сообщения up, ready и т. д. Для объектов класса clock это сообщения reset, time, wait и т. д. Так происходит потому, что каждый класс обладает набором функций, которые связаны с объектами класса. Функции являются частью этого класса объектов — его членами. На рис. 16.1 показан объект, содержащий функции-члены. Программа посылает этому объекту *сообщения* (messages), которые вызывают *функции-члены* (member functions) данного объекта. Затем эти функции-члены соответствующим образом обрабатывают объект.

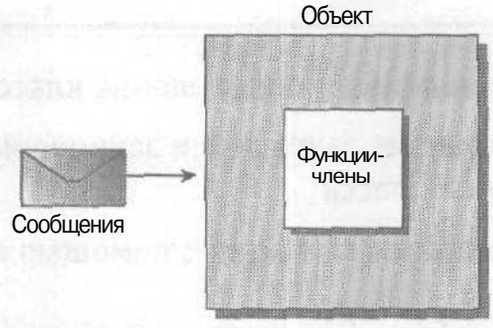


Рис. 16.1. *Сообщения объекту, в котором имеются функции-члены*

Эти функции называются *функциями-членами,* поскольку принадлежат классу, то есть являются его *членами.* Функции-члены программируются так же, как обычные функции — *функции-не-члены.* Однако, как мы увидим позже, функции-члены объявляются в классе и могут использоваться только с объектами этого класса.

Теперь вы можете лучше понять связь между классом и объектом. Понятие *класс* (class) относится ко всем объектам, которые ведут себя одинаково. Например, все окружности имеют вполне определенную форму, они обладают такими атрибутами, как местоположение, цвет и диаметр. *Объект* (object) — это конкретный экземпляр данного класса. Например, Земля имеет размер, цвет и местоположение, отличные от аналогичных параметров для Луны или Солнца. Связь между классом и объектами в сущности такая же, как между типом и переменными этого типа.

На уроке 16 вы научитесь создавать собственные классы и поймете, как работают классы. В большинстве примеров используются экранные объекты, так что вы сможете наблюдать интересные приложения классов.

Создание классов

Гимнасты (объекты класса athlete), которых вы создавали и с которыми работали на протяжении всей книги, имеют следующие функции-члены:

Oready()

O ир()

0 left()

Oright()

Они могут использоваться только с объектами класса athlete. Это означает, что мы можем записать инструкцию:

```
Sal.ready();
```

С другой стороны, если у нас нет другой функции ready(), не являющейся функцией-членом, то следующая инструкция недопустима:

```
ready();
```

Кроме того, поскольку в классе Clock нет функции-члена readyQ, недопустима и такая инструкция:

myclock. ready();

Поскольку до сих пор мы обрабатывали объекты, вызывая их функции-члены, я надеюсь, вы достаточно хорошо поняли, что собой представляют эти функции-члены. На рис. 16.2 показан класс athlete и его функции-члены.



Рис, 16.2. Функции-члены в объектах класса athlete

Другой очень важный аспект объектов, возможно, до сих пор оставался незамеченным. Как правило, каждый объект характеризуется некоторыми данными, определяющими его состояние. Это не относится исключительно к объектам, которые рассматриваются в данной книге. В любом классе можно определить объекты так, чтобы в них входили не только функции, но и данные.

Например, гимнасты появляются в некотором месте экрана. Это место идентифицируют координаты *x* и *y*. Значит, в каждом таком объекте должны быть две переменные для хранения значений координат. Аналогично, форма гимнаста определяется набором прямоугольников и окружностей. Каждый прямоугольник представляет собой объект класса Square (квадрат, вытянутый до прямоугольника), а окружность — объект класса Circle. Таким образом, помимо переменных x и y, каждый объект в качестве данных-членов должен содержать еще одну окружность класса Circle и пять квадратов класса Square, имеющих соответствующие размеры и положения.

В другом примере, который мы разберем более подробно, часы имеют только одну переменную в качестве данных-членов. Поскольку объекты класса Clock предназначены для хранения интервала времени, прошедшего с некоторого момента, то должен быть способ узнать, когда объект был создан или инициализирован. Это делается с помощью переменной timestarted, в которой хранится системное время (показания встроенных в компьютер часов), прошедшее с момента создания или инициализации объекта. Функция-член time() просто вычитает значение переменной timestarted из текущего значения системного времени, а функция-член reset() просто копирует текущее значение системного времени в переменную timestarted.

Данные-члены могут быть разных типов. Это могут быть встроенные типы, например int, float, long и т. д. Кроме того, это могут быть пользовательские типы, например athlete, Square, Circle, и другие типы любых доступных вам классов.

Каждый объект данного класса обладает полным набором функций-членов и данных-членов, определенных для этого класса. Например, невозможно, чтобы один объект класса Circle (например, Земля) имел цвет, в то время как другой (например, Солнце) его не имел. Однако в данных-членах скорее всего будут храниться разные значения для разных объектов (так, каждый гимнаст имеет свои координаты, а каждая планета — свои размеры). Это означает, что каждый объект будет иметь собственный набор данных. В отличие от данных-членов все функции-члены одинаковы, поэтому одна и та же функция может использоваться со всеми объектами данного класса. Последнее означает, что в программу включается только одна копия функции-члена данного класса.

Например, если вы объявите два объекта clock1 и clock2 класса Clock, то необходимо, чтобы в каждом из них можно было хранить разные значения переменной timestarted, поскольку они могли быть созданы или инициализированы в разные моменты времени. Таким образом, если вы объявите тысячу объектов класса Clock, то у вас будет тысяча переменных timestarted — по одной на каждый объект, — но только по одной копии функций reset(), time() и wait().

Составляющие класса объектов

Вам наверняка хочется понять, из чего строится класс объектов. Давайте рассмотрим, как построен класс Clock.

В обычных часах время показано на циферблате. Программные объекты класса Clock работают несколько иначе:

- Они могут сообщать время, прошедшее с момента создания или инициализации объекта.
- Значение времени хранится только в секундах (а не в часах, минутах и секундах).
- Они могут выполнять операции reset(), time(), wait() и watch().
- Им нельзя присвоить заданное значение. Их можно только инициализировать (перевести в исходное состояние).



ПРИМЕЧАНИЕ В объектах класса Clock, с которыми вы будете работать в примерах и упражнениях, используются системные (встроенные в компьютер) часы.

Программный код членов класса Clock

Операционная система Windows предоставляет функцию для определения времени, прошедшего с момента включения компьютера:

```
timeGetTime();
```

Если вы хотите использовать непосредственно эту функцию, то должны включить в программу заголовочный файл mmsystem.h.

Эта функция возвращает длинное целое, равное числу миллисекунд, прошедших с момента запуска системы. Если мы сохраним это значение в переменной timestarted, то сможем определить функции-члены класса Clock следующим образом:

```
void reset()
{
   timestarted=timeGetTime();
}
float time()
{
   // Чтобы результат измерялся в секундах, а не в миллисекундах,
   // его нужно разделить на 1000 (или умножить на 0.001)
   return(timeGetTime()-timestarted)*.001;
}
```

Функция wait() тоже довольно тривиальна. Если нам необходимо сделать паузу на определенное число секунд (которое задается в переменной tsec), то требуемое значение можно получить, добавив текущее время к значению переменной tsec.

Мы можем назвать этот результат willbe. Остается только организовать цикл и ждать, пока системное время не сравняется со значением переменной willbe.

```
void wait(float tsec)
{                              // Если нужна пауза в tsec секунд,
   float willbe;               // то нам придется ждать,
   willbe=time()+tsec;         // пока текущее время не станет равным
   while(time()<willbe;        // времени вызова функции плюс tsec
}
```

Обратите внимание, что функция-член time() оказалась внутри другой функции-члена waitQ. Это совершенно нормально, но один факт все же может показаться вам необычным. Вы уже должны были привыкнуть, что функция-член класса всегда вызывается вместе с объектом этого класса. Здесь же функция-член time() вызывается сама по себе, как самая обычная функция.

Неявные ссылки на объекты

Такой вызов функции-члена time() возможен потому, что во время работы программы функция wait() используется с тем же объектом класса Clock, что и функция time().



ПРИМЕЧАНИЕ Предполагается, что функции-члены оперируют с тем объектом, который в данный момент используется.

Например, пусть объект BigBen объявлен как экземпляр класса Clock:

```
Clock BigBen;
```

Далее, пусть функция wait() вызывается с объектом BigBen:

```
BigBen.wait(10);
```

Тогда, если функция waitQ вызывает функцию time(), по умолчанию предполагается, что функция time() тоже используется с объектом BigBen.

И никак иначе программа работать не будет. Когда вы составляете код функции-члена (такой, как wait() или timeQ), то еще не знаете, с каким именно объектом она будет вызвана. Будет ли этим объектом mywatch? Или timer? А может быть BigBen? Внутри функции нельзя заранее указать имя объекта, поскольку до вызова функции оно неизвестно.

Вы можете быть уверены только в одном — что это объект класса Clock, в котором имеются переменная timestarted типа long и функции-члены waitQ, timeQ и reset(). Когда вызывается функция-член, компилятор запоминает, с каким объектом она вызвана, и использует данные этого объекта в тех вызовах функций-членов, в которых опущено имя объекта.

В представленном ниже программном коде функции-члена wait() максимально возможное время паузы ограничено значением 60 секунд:

```
void wait(float tsec)
{
   float willbe;
   if(tsec>60) tsec=60;      // Ограничение максимального времени
                             // паузы одной минутой
   willbe=time()+tsec;
   while(time()<willbe);
}
```

ВНИМАНИЕ Обратите внимание, что цикл while всего лишь сравнивает значения текущего времени и переменной willbe. Он продолжается до тех пор, пока возвращаемое значение функции time() не окажется больше значения переменной willbe. Все это время компьютер ничего не делает. Для многозадачных компьютеров это не очень хорошо.

Объявление класса объектов

Поскольку вы уже знаете, как писать функции, то могли бы и сами написать коды для функций reset(), time() и wait(). Мы пока не знаем, как объединить эти функции в один класс, но тем не менее очень близки к этому.

Класс создается в два этапа:

- В *объявлении класса* (class declaration) ему назначается имя, а также описываются все его переменные или объекты (данные-члены) и все его функции (функции-члены). Программный код функций-членов в объявление класса не входит.
- *Определение класса* (class definition) включает в себя код всех его функций-членов.

Обычный формат объявления класса имеет следующий вид:

```
class имя_класса
{
   объявление данных-членов
   объявление функций-членов
};                          // Не забывайте про точку с запятой!
```

ПРИМЕЧАНИЕ Классы могут объявляться как структуры, если вместо ключевого слова class использовать ключевое слово struct. С этим приемом вы познакомитесь на уроке 20, однако применять его не рекомендуется — он не дает абсолютно никаких преимуществ.

Объявление данных-членов и функций-членов должно быть в фигурных скобках, за которыми нужно обязательно ставить точку с запятой! Пропуск точки с запятой представляет собой весьма распространенную ошибку. Выдаваемые в ответ на эту ошибку сообщения компилятора очень туманны и вряд ли помогут вам ее обнаружить.

Приведенный основной формат объявления класса должен содержать в себе следующие пункты:

- *Имя класса* — любое допустимое имя.
- *Объявление данных-членов* — объявление всех переменных или объектов, которые будут принадлежать каждому объекту этого класса.
- *Объявление функций-членов* — список *прототипов* (prototypes) всех функций-членов. Прототип идентифицирует функцию, тип ее возвращаемого значения, а также число и типы ее аргументов. После прототипа ставится точка с запятой. Объявления данных-членов и функций-членов могут располагаться в любом порядке.

Класс Clock в качестве примера

В качестве примера давайте рассмотрим одно из возможных объявлений класса Clock.

```
class Clock//**********************Clock
{
   long timestarted;
public:
   float time();              // Функция, возвращающая истекшее время
   void wait(float tsec);     // Функция задания паузы в tsec секунд
   void reset();              // Функция перезапуска
};
```

В первой строке ставится ключевое слово class и идентификатор (имя) нового класса, который мы создаем. Следующая далее последовательность инструкций, заключенная в фигурные скобки (после которых стоит точка с запятой), представляет собой объявление класса.

В списке данных-членов класса Clock только одна переменная timestarted типа Long. Она объявляется непосредственно после открывающей скобки, но может быть объявлена в любом другом месте внутри объявления класса.

Затем перечисляются прототипы функций-членов: time(), wait() и reset(). Отметьте, что каждый прототип по существу повторяет первую строку самой функции. Каждый прототип содержит тип возвращаемого значения, идентификатор (имя) и в скобках типы аргументов функции. Имена аргументов в прототипы включать не обязательно (в данном случае можно опустить имя переменной tsec в прототипе функции wait()).

Единственное, чего мы здесь не знаем, это назначение слова public, с которым мы познакомимся несколько позже.

Определение класса

Как уже говорилось, определение класса содержит в себе полный программный код всех его функций-членов. При этом существенно, что код функции-члена, как и код обычной функции, включается в программу (а не в объявление класса).

Как же обозначить, что функция является членом того или иного класса. В принципе язык C++ позволяет вам иметь обычную функцию wait() в дополнение к функции wait() — члену класса Clock.

Благодаря перегрузке функций несколько функций могут иметь одно и то же имя. Следовательно, необходимо четко обозначать, что код, который вы создаете, относится к функции-члену данного класса.

Определяя функцию, вы должны связать ее имя с ее классом. Для этого имя функции уточняется. *Уточнение* (qualification) имени функции представляет собой имя класса, за которым следуют два двоеточия. Синтаксис уточнения имени функции:

```
имя_класса :: имя_функции (список_аргументов)
```

Б нашем примере с часами уточнение имени функции будет выглядеть следующим образом:

```
Clock :: walt(float tsec)
```

Теперь мы можем написать полное определение класса Clock:

```
void Clock :: reset()
{
   timerstarted=timeGetTime();
}
float Clock :: time()
{
   // Чтобы результат измерялся в секундах, а не в миллисекундах,
   // его нужно разделить на 1000 (или умножить на 0.001)
   return(timeGetTime()-timestarted)*.001;
}
void Clock :: wait(float tsec)
{
   float willbe;
   if(tsec>60) tsec=60;     // Ограничение максимального времени
                            // паузы одной минутой
   willbe=time()+tsec;
   while(time()<willbe);
}
```



СОВЕТ Вы можете попробовать написать программный код этого класса и использовать его в своей программе. Однако поскольку в вашем программном обеспечении уже есть класс Clock, то компилятор выдаст сообщение об ошибке. Избежать этого можно, заменив имя Clock именем clock. Кроме того, вам необходимо включить в программу заголовочный файл mmsystem.h.

Открытые, закрытые и защищенные члены класса

При создании класса может возникнуть необходимость, чтобы члены класса (данные или функции) были либо видимы, либо не видимы для других частей программы. Здесь имеются три возможности.

Открытые члены

Доступ к *открытым* (public) членам класса могут иметь не только функции-члены класса, но также и любая часть программы, в которой имеется доступ к объекту. В классе Clock открытыми членами являются функции reset(), time() и wait(). Таким образом, даже вне класса Clock эти функции могут работать с объектами этого класса.

Закрытые члены

Доступ к *закрытым* (private) членам класса могут иметь только функции-члены этого класса. Доступ к ним из других частей программы невозможен. По умолчанию все члены класса считаются закрытыми. Таким образом, если ничего дополнительно не указано, член класса является закрытым. Переменная timestarted в нашем примере является закрытой, поэтому эту переменную могут использовать только функции reset(), time() и wait(), которые являются членами класса Clock.



ПРИМЕЧАНИЕ Способность изолировать информацию от внешнего вмешательства в объектно-ориентированном программировании называется *инкапсуляцией*. Благодаря тому, что все манипуляции с данными могут осуществляться только ассоциированными с ними функциями-членами (то есть предусмотренными для этого инструментами), программное обеспечение становится более надежным. Кроме того, его легче поддерживать, поскольку все модификации обычно ограничиваются только теми объектами, которые в них нуждаются, и не распространяются на остальную часть программного обеспечения.

Защищенные члены

Доступ к защищенным (protected) членам класса могут иметь не только функции-члены класса, но также и функции-члены производных классов, с которыми мы познакомимся немного позднее. Ключевое слово public, private или protected сохра-

няет свое действие, пока в объявлении класса не появится следующее ключевое слово. Другими словами, при объявлении класса изначально по умолчанию подразумевается ключевое слово private. Пока не появится слово public или protected, все члены считаются закрытыми. Затем в силу вступает новое ключевое слово, до тех пор пока не появится следующее. В примере с классом Clock все функции-члены являются открытыми, поскольку перед первой из них стоит ключевое слово public, а слов private и protected нет в объявлении класса.

Конструкторы

Как вы знаете, объект класса Clock при объявлении автоматически инициализируется. Это означает, что при создании нового объекта класса Clock переменной timestarted присваивается текущее системное время. Кто (или вернее что) это делает?

Мы должны научиться определять специальную функцию, которая будет автоматически вызываться при создании каждого объекта. В языке C++ это можно сделать при помощи специальной функций, которая называется *конструктором* (constructor).

Конструктор похож на любую другую функцию-член, за исключением следующего:

- Имя конструктора совпадает с именем класса. Например, конструктором класса Clock является функция Clock().
- При создании нового объекта конструктор вызывается автоматически. Например, если вы создаете два объекта mine и yours класса Clock, то конструктор ClockQ будет вызван дважды — один раз при создании объекта mine и другой при создании объекта yours.
- Конструктор нельзя вызвать из программы напрямую. Например, нельзя написать инструкцию mine.Clock();. Конструктор вызывается только однажды — при создании объекта.
- У конструктора нет возвращаемого типа (даже типа void). Возможно существование нескольких конструкторов с разными списками аргументов. Например, в классе Box есть конструктор без аргументов и конструктор с одним аргументом. Так, инструкция box x, y("Y:"); вызывает конструктор без аргумента для объекта x и конструктор с одним аргументом для объекта y. В этом случае в информационной рамке объекта y автоматически появится метка Y:.



ПРИМЕЧАНИЕ Конструкторы должны быть открытыми. К ним необходима возможность доступа из любой части программы, в которой объявляется объект данного класса, в противном случае при создании объекта автоматическая инициализация станет невозможной.

Инициализация объектов

Конструкторы исключительно полезны для инициализации объектов. Они дают возможность подготовить объект к использованию. Конструкторы имеются в

большинстве классов, с которыми вы уже работали. Часы подводятся, и гимнасты становятся один за другим. Другим экранным объектам присваиваются координаты по умолчанию и прочие атрибуты.

Полное объявление класса Clock имеет следующий вид:

```
class Clock //**************************Clock
{
protected;
   long timestarted;
publ ic:
   Clock();
   float time();              // Функция, возвращающая истекшее время
   void wait(float tsec);    // Функция задания паузы в tsec секунд
   void watch(float tsec);   // Функция задания паузы,
                             // пока время не станет равным tsec
   void reset();             // Функция перезапуска
};
```

Кроме того, изменится определение класса. В нем появится код конструктора:

```
Clock : : Clock()
{
   tircestarted=timeGetTime();
}
```

Использование объектов

При создании класса вы не создаете ни одного объекта этого класса. Вы лишь описываете поведение будущих объектов класса. Даже если вас интересует только один экземпляр часов, вы все равно должны предоставить описание поведения всех возможных часов. Объект класса Clock можно объявить в программе только после того, как создан класс Clock.

Чтобы использовать объект, его нужно только объявить. Это делается точно так же, как вы объявляли гимнастов и часы. Объявление состоит из имени класса, за которым через пробел следуют идентификаторы (имена) необходимых объектов. Если вам нужно несколько объектов, то их идентификаторы разделяются запятыми;

```
имя_класса идентификатор_объекта_1, идентификатор_объекта_2, . . . ;
```

В наших примерах объявления объектов происходили следующим образом:

```
Clock timer, mywatch;
athlete Sal;
```

Объявление объекта (или объектов) сообщает компилятору, что вы будете использовать в своей программе один или несколько объектов данного класса и обра-

щаться к ним по указанным идентификаторам. Затем компилятор выделяет место в памяти компьютера, после чего можно быть уверенным, что, когда процесс выполнения программы дойдет до этого места, будет вызван подходящий конструктор для инициализации объекта.

Обратите внимание, как объявление класса похоже на объявление переменной:

```
тип идентификатор_переменной_1, идентификатор_переменной_2, ...;
```

Например:

```
float x, y;
```

Это еще одно доказательство того, что между типами и классами имеется очень большое сходство. На этом уроке вы также узнаете, что переменные представляют собой простейшие формы объектов, поэтому очень часто те и другие мы называем одним словом — объектами.

Доступ к членам объекта

Доступ к открытым членам (функциям или данным) возможен из любой части программы. При таком доступе после имени объекта через точку ставится имя члена. Именно так осуществлялся доступ к функциям-членам класса athlete. Например, следующая инструкция вызывает функцию ready(), являющуюся членом объекта sal:

```
sal.ready();
```

То же самое можно делать и с данными-членами. Например, переменная timestaeted относится к данным-членам класса Clock, и для доступа к ней перед именем переменной нужно указать имя объекта:

```
bigben.timestarted = mine.timestarted;
```

Тем не менее поскольку переменная timestarted является закрытой, то к ней нет доступа из других частей программы, а только из функций-членов класса Clock.

Доступ к **членам** в функциях-членах

По коду функции-члена компилятор понимает, как нужно работать с каждым объектом созданного вами класса. Поскольку функции-члены могут вызываться только для объектов данного класса, то компилятор автоматически понимает, что любая ссылка на функции-члены или на данные-члены внутри определенной функции-члена относится к объекту, для которого она вызвана. Бот почему внутри функций-членов нет необходимости уточнять имена данных-членов и функций-членов, указывая перед их именами имя объекта.

Создание нового класса объектов

Теперь мы готовы экспериментировать с новым классом — классом, который мы создадим сами. Это класс wagon (вагон). Вагоны можно будет изображать на экра-

не и перемещать в горизонтальном направлении. Наш вагон состоит из прямоугольника и двух окружностей, как показано на рис. 16.3.



Рис. 16.3. *Вагон*

Класс wagon будет иметь конструктор и функцию-член move(), которая заставит вагон перемещаться по экрану. Поскольку вагон будет двигаться, то нам, очевидно, потребуется пара переменных, чтобы контролировать его положение на экране.

Вот одно из возможных объявлений класса wagon:

```
class wagon
{
   Circle frontwheel, rearwheell;
   Square body;
   float x, y;          // Координаты
public:
   wagon();
   void move();
};                      // Не забывайте про точку с запятой!
```

Здесь три объекта играют роль данных-членов: frontwheel, rearwheell (две окружности), body (квадрат) и две переменные с плавающей точкой x и y. Поскольку никак не обозначено, что все эти члены открыты, то они будут считаться закрытыми. Это удобно, поскольку иметь к ним доступ будут только функция-член move() и конструктор.

Создание вагона с помощью конструктора

Мы уже знаем, что конструктор будет вызываться для каждого создаваемого объекта. Конструктор присвоит подходящий размер, цвет и положение объектам, из которых состоит вагон, а также установит начальные значения координат x и y. Ниже представлено определение класса wagon (его функции-члены):

```
const int red=1, green=2;
wagon :: wagon()
{
   // Начальные координаты:
   x=60;
   y=200;
   // Размер колес:
   frontwheel.resize(20);
```

```
    rearwheel.resize(20);
    // Размер корпуса:
    body.resize(20, 80);
    // Цвет:
    frontwheel.color(red);
    rearwheel.color(red);
    body.color(green);
    // Размещение всех частей в заданных местах:
    body.place(x, y);
    rearwheel.place(x-25,y+10);
    frontwheel.place(x+25,y+10);
    // Отображение всех частей на экране:
    body.show();
    rearwheel.show();
    frontwheel.show();
}
void wagon :: move()
{
    // Удаление всех частей с прежнего места экрана:
    body.erase();
    rearwheel.erase();
    frontwheel.erase();
    x++;                          // Инкремент x
    // Размещение всех частей на новом месте;
    body.place(x,y);
    rearwheel.place(x-25,y+10);
    frontwheel.place(x+25,y+10);
    // Отображение всех частей на новом месте:
    body.show();
    rearwheel.show();
    frontwheel.show();
}
```

Представленный выше класс wagon теперь вполне работоспособен, и вы можете в этом убедиться. Теперь было бы неплохо определить функции erase(), place() и show(), но есть способ лучше.

Создание вагона с помощью класса Stage

С помощью знакомого нам класса Stage можно вставить все детали вагона в один объект, что значительно упрощает программу. Это видно из представленной ниже программы c6train.cpp:

```
ffinclude"franca.h"
// Реализация класса wagon          c6train.cpp
/,/ Версия 2.0, с использованием класса Stage
//
int const red=1, green=2;
class wagon
{
   Stage railroad;                  // Объявление объекта класса Stage
   Circle frontwheel, rearwheel;    // Объявление колес и корпуса
   Square body;
   float x, y;                      // Объявление координат
public:
   wagon();
   void move();
}; // Не забывайте про точку с запятой!
wagon : : wagon()
{
   x=60;                            // Начальные координаты
   y=200;
   railroad.place(x,y);
   frontwheel.resize(20);           // Размер объектов
   rearwheel.resize(20);
   body.resize(20, 80);
   frontwheel.color(red);           // Цвет объектов
   rearwheel.color(red);
   body.color(green);
   body.place(x,y);                 // Размещение объектов
   rearwheel.place(x-25,y+10);
   frontwheel.place(x+25,y+10);
   // Вставка объектов в объект класса Stage
   railroad.insert(rearwhee );
   railroad.insert(frontwheel);
   railroad.insert(body);
}
void wagon : : move()
{
   // Удаление объекта с прежнего места экрана
   railroad.erase();
   x++;                          // Шаг вперед
   railroad.place(x,y);          // Размещение на новом месте
```

```
    railroad.show();          // Отображение в текущей позиции
}
```

В новой версии программы определение класса изменено. Теперь там остался только один объект класса Stage. Новый конструктор не слишком изменился, за исключением того, что теперь он просто вставляет детали вагона в объект класса Stage, а не отображает их на экране.

Испытание вагона

Функция move() стала проще, поскольку теперь все операции выполняются над одним объектом класса Stage, а не повторяются с каждой деталью конструкции вагона.

В следующей программе вы можете испытать свой новый класс:

```
void mainprog()
{
    int i;
    wagon front;              // Объявление первого вагона, при этом
                              // вызывается конструктор класса wagon
    Clock station;            // Объявление другого объекта, и теперь
                              // вызывается конструктор класса Clock
    for(i=1; i<=80; i++);     // Цикл
    {
        front.move();         // Перемещение вагона
        station.wait(.033);
    }
    wagon caboose;            // Объявление второго вагона. Снова
                              // вызывается конструктор класса wagon.
                              // Новый объект инициализируется только
                              // после окончания предыдущего цикла
    for(i=1;i<=200; i++);     // Цикл
    {
        front.move();         // Перемещение обоих вагонов
        caboose.move();
        station.wait(.06);
    }
}
```

Доступ к членам класса

С самого начала этой книги мы работали с функциям-членам класса athlete. Для доступа к функции-члену перед именем функции указывается имя объекта, за которым ставится точка. Например:

```
Sal.ready();
```

Эта инструкция вызывает функцию-член ready() объекта SaL, который принадлежит классу athlete. Тем же способом вызываются данные-члены. Например, в программе c6train.cpp, в которой используется класс wagon, можно следующим образом попытаться получить доступ к данным-членам этого класса:

```
front.x=0;                    // Задание координаты x первого вагона
front, rearwheel.resize(15);  // Изменение размера заднего колеса
cabooze.y=10;                 // Задание координаты y второго вагона
```

Однако при такой попытке доступа компилятор выдаст сообщение об ошибке, что эти данные-члены недоступны. Почему? Да потому, что все данные-члены класса wagon по умолчанию считаются закрытыми. Если перед объявлением данных-членов вставить ключевое слово public (за которым ставится двоеточие), можно попробовать повторить попытку.

В данном примере вагон создается всегда в точке с координатами (60, 200). Возможно, стоит попробовать разместить вагон где-нибудь в другом месте или позволить пользователю самому определять координаты точки. Рассмотрим несколько вариантов решения проблемы:

- Сделать доступ к координатам вагона открытым.
- Использовать конструктор с параметрами.
- Иметь функцию-член для изменения координат.

Открытый доступ к координатам

Если сделать координаты открытыми, после создания объекта их можно будет изменить. Следовательно, вагон можно будет изображать в разных положениях. На первый взгляд кажется неплохо. Так ли это? Вы еще получите возможность убедиться, что это не так!

Этот вариант легко осуществить — достаточно перед объявлением переменных x и y включить в объявление класса wagon следующую инструкцию:

```
publ ic:
```

Теперь **можно изменять положение** каждого вагона, **напрямую** меняя **значения его координат. Например:**

```
front.x=120;
front.y=100;
caboose.x=60;
caboose.y=100;
```

Такие инструкции будут работать только в том случае, если изменять каждую координату *перед* перемещением вагона. Например:

```
front.x=180;
front.moveO;
```

Однако, если изменять координаты после того, как перемещение вагона началось (он уже появился на экране), можно получить странные результаты. Например:

```
front.move();
front.x=200;
front.move();
```

Подумайте, что произойдет в этом случае, а затем измените программу и посмотрите на результат. Он вам нравится? Понимаете, что произошло?

Было бы еще хуже, если бы этим координатам где-нибудь в программе были присвоены непредусмотренные вами значения:

```
caboose.x=z1;
caboose.y=z2;
```

Вы можете спросить, что же здесь неправильного? В самом деле, ничего — разве что переменные z1 и z2 получены непредусмотренным вами образом (например, в результате каких-то сложных вычислений они оказываются равными -432 и 221.345).

Б обоих случаях мы получаем нежелательный результат. В первом случае функция-член move() удаляет текущее изображение вагона, перемещает его на один шаг по горизонтали и задает рисует его новое изображение. Если изменить координаты, то следующая функция move() удалит (вернее закрасит белым цветом) не вагон, а заданное новыми координатами место экрана, поскольку функция erase() удаляет объект, используя его текущие координаты!

Вот почему в вашем классе нужны закрытые или защищенные члены. Доступ к этим данным-членам могут получить только функции-члены, а поскольку вы сами пишете и тестируете эти функции, вы и обеспечиваете их работоспособность.

Можно, конечно, возразить. Поскольку мы сами создаем программу, то должны знать, что в ней происходит с этими данными-членами. Но лучше не переоценивать свою способность полностью контролировать ситуацию в программе (особенно большой и сложной).



СОВЕТ Бит профилактики стоит мегабайта лечения...

Наконец, главный аргумент в пользу того, чтобы не открывать большую часть данных-членов, состоит в том, что никогда нельзя писать программы только для себя. И уж тем более никогда не считайте, что вы создадите программу, поработаете с ней, а затем выбросите. Программы имеют тенденцию использоваться снова и снова. А если вы станете профессиональным программистом, работоспособность ваших программ будут поддерживать совсем другие люди. Кроме того, программы могут создаваться целыми коллективами, и на основе созданного вами кто-то другой будет строить нечто большее. Любая возможность избежать ошибок должна быть принята во внимание.

Конструктор с параметрами

Другой вариант решения проблемы задания координат вагона состоит в том, чтобы позволить конструктору получать значения координат в качестве аргументов. Например:

```
wagon : : wagon( int x0, int y0)
{
   x=x0;
   y=y0;
   ...
```

Если изменить таким образом конструктор (не забудьте еще изменить объявление конструктора в объявлении класса), то вагоны можно будет размещать в любом выбранном вами место.

```
wagon front (120, 220);
...
wagon caboose (60, 220);
```

Конструктор без параметров называется также *конструктором по умолчанию* (default constructor). Я рекомендую вам всегда включать в ваш класс конструктор по умолчанию, даже при наличии конструктора с параметрами.

Использование функции-члена

Возможен еще один вариант решения проблемы — включить в класс функцию-член для размещения вагона в заданном положении. Это вполне разумно и гораздо лучше, чем делать координаты открытыми. Функция-член могла бы при этом проверять допустимость значений заданных координат.



СОВЕТ

Этот третий вариант реализации класса wagon имеет некоторые преимущества перед вторым. Однако и он недостаточно хорош! Изучив наследование на уроке 17, попробуйте решить эту проблему, наследуя класс wagon от класса Stage, и вы поймете, насколько это лучше.

Самостоятельная практика

- Модифицируйте класс wagon так, чтобы он изображал паровоз. Для имитации кабины и трубы добавьте туда пару прямоугольников.
- Переделайте класс Clock в новый класс (например, Clockl) так, чтобы функция-член mark() автоматически обнуляла показания часов через заданный промежу-

ток времени. Другими словами, если вы захотите организовать пятисекундные паузы, то могли бы воспользоваться следующими инструкциями:

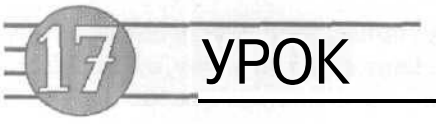
```
Clock mytimer;
for(int i=1, i<=20, i++);
{
   mytimer.mark(5.);
}
```

- Разработайте класс pair для создания двух гимнастов. В классе должны быть функции-члены ready(), up(), left() и right(). Напишите программу, в которой эта пара гимнастов (два объекта класса pair) будет выполнять подскоки.
- Создавая функцию-член Clock::wait(float), для хранения текущего времени я использовал переменную willbe. Попробуйте для получения времени воспользоваться функцией Clock::time().
- Уберите точку с запятой после закрывающей скобки в объявлении класса wago и попытайтесь откомпилировать программу. Изучите сообщение об ошибке.

Что нового мы узнали?

В этом уроке мы научились

- Путем объявления и определения класса строить новый класс.
- Выбирать между открытыми, закрытыми и защищенными членами класса.
- Для инициализации объектов создавать конструкторы.
- Создавать новый класс объектов.
- Через объекты обеспечивать доступ к данным-членам и функциям-членам класса,



Производные классы

- Объявление и определение производных классов
- Наследование членов класса
- Полиморфизм

Создание собственных классов объектов — это не самое интересное приложение классов. Гораздо удобнее получить новый класс из уже существующего. При этом не нужно переделывать все, что вы наследуете. Достаточно изменить только то, что вам не подходит.

Мы говорим, что объекты являются разумными, поскольку они знают, как выполнять наши команды. Например, объекты класса athlete умеют занимать позиции ready, up, left и right. Они также могут «говорить» некоторые вещи. На этом уроке вы научитесь создавать еще более умные объекты.

На предыдущем уроке в нескольких упражнениях раздела «Самостоятельная практика» я просил вас модифицировать класс wagon. Чтобы успешно выполнить эти задания, нужно было выбрать один из следующих вариантов действий:

- Взять старый код, изъять из него те фрагменты, которые больше не нужны, и написать вместо них новые. В этом случае у вас останется только один вариант кода для нового класса. Прежнего класса не станет.
- Сделать копию старого кода и сделать в ней необходимые изменения. В этом случае вы получаете две версии одного класса. Можно даже дать новому классу другое имя. В программе можно будет использовать объекты обоих классов.

Второй вариант несколько предпочтительней, поскольку старый класс остается неизменным и готовым к работе. К тому же в нем нет новых ошибок. Однако теперь при появлении проблем вам придется проверять обе части программы! Кроме того, используя два схожих класса в одной программе, вы дублируете сохранившиеся в обоих классах (не изменившиеся) фрагменты кода.

Два (или более) фрагмента кода, делающие абсолютно одно и то же, — это большой недостаток программы. Пока вы, возможно, этого не понимаете, но поскольку всякая программа по разным причинам (не только из-за ошибок) обычно нуждается в доработке, одни и те же изменения в будущем придется вносить во все одинаковые фрагменты кода. Поэтому гораздо лучше несколько раз использовать один фрагмент.

Получение классов из существующих классов

Объектно-ориентированное программирование обеспечивает возможность многократного использования вашего программного кода. Если вам нужен класс объектов, очень похожих на существующие, но с несколько отличными свойствами, можно построить новый класс из уже существующего. В C++ исходный класс называется *базовым классом* (base class), а тот, который вы из него создаете, — *производным классом* (derived class).

Особый сорт вагона, похожего на паровоз, и особый сорт часов, обнуляющих свои показания через заданный промежуток времени, — это прекрасный пример произ-

водных классов. Мы переходим к новым классам: классу locomotive, представляющему собой специализированный вариант класса wagon, и классу timer, представляющему собой специализированный вариант класса Clock.

Каждый объект производного класса может одновременно считаться объектом соответствующего базового класса. Паровоз остается вагоном, а секундомер — часами.

Производные классы обладают очень полезным свойством: все, что уже работает в базовом классе, будет автоматически *наследоваться* (inherited) производным классом.

Например, чтобы создать класс locomotive, не нужно описывать, как изобразить колеса вагона или как он двигается. Единственное, что вы должны сделать, — это описать, как изобразить трубу и кабину, поскольку их не было в классе wagon.

Точно так же для класса timer вы должны написать только функцию-член mark(), которая будет обнулять показания часов через заданный промежуток времени.

Создание производных классов

Создание одного класса из другого включает в себя этапы, очень напоминающие создание нового класса:

1. Объявить класс — объявить все члены, которых нет в базовом классе (вы должны также объявить функции-члены, которые заменяют функции-члены, существующие в базовом классе). При объявлении производного класса явно указывается, что этот класс является производным от данного базового класса.
2. Определить класс — написать код для новых функций-членов.

Объявление производного класса

При объявлении производного класса необходимо, во-первых, указать, что это производный класс, а во-вторых, указать соответствующий базовый класс. Синтаксис объявления производного класса следующий:

```
class имя_производного_класса: public имя_базового_класса
{
   объявления членов класса
};                          // Не забывайте про точку с запятой!
```

Обратите внимание на двоеточие между ключевым словом public и именем производного класса, которое позволяет идентифицировать базовый класс для объявляемого производного класса.

В следующем фрагменте показан возможный вариант объявления класса timer:

```
class timer:public Clock
{
   void mark(float time);
};
```

Определение **производного** класса

Определение новой функции-члена может быть следующим:

```
void timer::mark(float time)
{
   watch(time);
   reset();
}
```

Объект нового класса timer по-прежнему может вызывать все функции-члены, существующие в базовом классе. Функции reset(), time(), watch() и wait() наследуются из базового класса. Про эти функции ничего не надо указывать дополнительно. Вы, фактически, можете даже не знать, как они работают! Это необычная черта наследования — взять уже готовое, над чем вы сами никогда не работали.

Классы и объекты в реальном мире

Классы и объекты объектно-ориентированного программирования представляют собой концепции, взятые из обыденной жизни. Объекты — это предметы, которые мы видим вокруг себя. Какие это объекты? Да все что угодно: любой из ваших коллег, деталь обстановки, машина на улице, светофор на перекрестке и даже птицы над головой! Все это примеры объектов (не забудьте еще про объекты, которые вы видите на экране компьютера).

Естественно, вы замечали, что некоторые из объектов похожи. Они могут одинаково выглядеть или иметь сходные характеристики. Возьмем, например, ваших коллег. Хотя каждый из них индивидуален, тем не менее все они обладают сходными характеристиками, отличающими их от светофоров, мебели или птиц.

Можно сказать, что объекты, имеющие сходные характеристики, относятся к одному классу. Попробуем идентифицировать несколько классов:

- Класс коллег.
- Класс светофоров.
- Класс мебели,
- Класс птиц.
- Класс экранных объектов.

Класс представляет собой набор всех объектов данного класса, а не один конкретный объект. Таким образом, хотя имеется класс светофоров, но светофор, стоящий около вашего дома, — это отдельный объект. Каждый ваш коллега тоже является отдельным объектом, как и стол, за которым вы сидите.

Более того, классы можно дробить и дальше, чтобы они точнее описывали мир. Например:

- Можно разделить класс *мебели* на столы, стулья, лампы, кровати и т. д.
- Можно классифицировать птиц по отдельным видам.
- Можно классифицировать ваших коллег по полу.

Родительские и дочерние классы

На рис. 17.1 показан класс *люди,* который разделяется на два класса: *мужчины* и *женщины.* В нижней строке показаны объекты этих классов.



Рис. 17.1. *Классы и объекты*

На этом рисунке содержится больше информации, чем это кажется на первый взгляд. Если вы внимательно на него посмотрите, то заметите, что из одной вершины все картинки опускаются вниз. Рисунок иллюстрирует класс *люди,* который определяет базовые атрибуты для всех остальных классов.

Класс *люди* является базовым для двух производных классов: *мужчины* и *женщины.* Не забывайте, что как *мужчины,* так и *женщины* остаются *людьми.* Оба класса имеют все базовые атрибуты класса *люди,* за исключением некоторых дополнительных атрибутов, характерных либо только для *мужчин,* либо только для *женщин.* Если сильно упростить реальную ситуацию, можно сказать, *что мужчин* отличают брюки, а *женщин* — юбка и длинные волосы.

Помните, что производный (или дочерний) класс имеет все атрибуты базового (или родительского класса). Следовательно, как уже говорилось, *мужчины* и *женщины* остаются *людьми.*

Обратите внимание, что производный класс представляет собой *специализированный класс* внутри заданного базового класса. *Мужчины* — это специализированная (по признаку пола) группа *людей* и *женщины* — это еще одна специализированная группа *людей*. Точно так же класс Circle представляет собой специализированный класс класса экранных объектов ScreenObj.

Объекты как экземпляры классов

Люди, мужчины, женщины — все это слишком неопределенно. Слишком многих можно обозначить этими именами классов. Если вы хотите обратиться к конкретному *человеку,* то должны указать на соответствующий конкретный *экземпляр* (instance) этого класса.

Возвращаясь к нашему простому примеру, в нижней части рисунка мы видим конкретных людей. Все они разные. Теперь мы можем говорить Билл, Джой, Дэн, Пол, Тереза, Юдит, Трейси, Андре. Билл — это не класс! Билл — это такой вот славный парень, которому повезло родиться мальчиком, то есть объектом класса *мужчины*. То же самое можно сказать и про Джоя, Дэна и Пола. А вот Тереза, Юдит, Трейси и Андре представляют собой объекты класса *женщины*. Но все эти объекты — это еще и объекты класса *люди*.

В своих программах вы также можете использовать объекты реального мира. Более того, вы уже работали с такими объектами: гимнастами, роботами, окружностями, квадратами, информационными рамками, часами.

Специализация экранных объектов

Идея создания одного класса из другого широко представлена среди экранных объектов. Класс экранных объектов ScreenObj позволяет размещать, удалять, изображать и раскрашивать объекты на экране, а также изменять их размеры. Класс Box является производным от класса ScreenObj. Другими словами, класс Box — это специализированный класс класса ScreenObj. Все, что можно делать с классом ScreenObj, можно делать и с классом Box. Но, кроме того, класс Box позволяет еще выводить сообщения на экран и помечать их меткой. Идея очень проста: если вы хотите создать *специализированный класс* из существующего класса, то должны сделать следующее:

- Дать новому классу имя и указать базовый класс. Например, если новым классом является Box, то необходимо указать, что он является производным от базового класса ScreenObj. Если новым классом является athlete, то необходимо указать, что он является производным от базового класса Stage.
- Объявить все переменные или объекты, которые будут в производном классе, но которых не было в базовом. Так, каждый объект класса Box может иметь данные, например метку и выводимое сообщение. Этих данных не было в классе ScreenObj. Каждый объект класса athlete будет иметь несколько других объектов: голову, тело, левую и правую руку, левую и правую ногу. И этих объектов нет в классе Stage,

- Объявить и написать код всех функций, существующих в производном классе и не существовавших в базовом. Так, каждый объект класса Box должен уметь обрабатывать сообщения Label и say, а каждый объект класса athlete — сообщения ready, up, left и right.

Объекты производного класса обладают всеми данными и функциями базового класса, равно как и новыми, добавленными при создании производного класса. Необходимо лишь написать программы для тех функций, которые мы добавляем. При этом мы можем пользоваться всеми существующими программами.

На рис. 17.2 показано соответствие между базовым классом ScreenObj и производными от него классами Circle и Box. Объект класса ScreenObj обладает данными и функциями, перечисленными во внутренних прямоугольниках. При этом объект класса Box или класса Circle обладает еще и дополнительными данными и функциями.

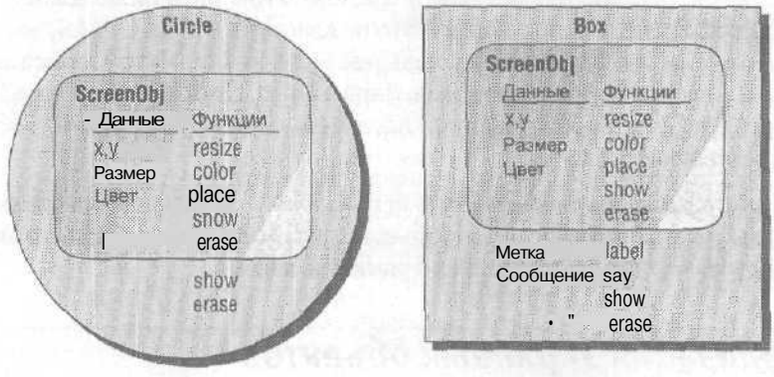


Рис. 17.2. *Класс ScreenObj и его производные классы Circle и Box*

Обратите внимание, что изображение объекта класса ScreenObj оказалось внутри изображения объекта класса Box. Таким образом обозначено, что объект класса Box, помимо собственных данных и функций-членов, содержит еще и те, которые унаследованы от класса ScreenObj. Аналогично, объект класса Circle помимо собственных данных и функций-членов, содержит еще и те, которые унаследованы от класса ScreenObj.

Наследование характеристик базовых классов

Отношения между классами можно представить иначе, например, с помощью иерархической диаграммы, показанной на рис. 17.3. На ней каждый блок представляет собой некоторый класс объектов. При этом базовые классы располагаются выше соответствующих производных классов. Например, класс ScreenObj является базовым для классов Circle, Square, Box и Stage, а класс Stage является базовым для класса athlete. Кроме того, хотя это и не показано, класс athlete является базовым для класса Robot (неудивительно, что они так похожи).

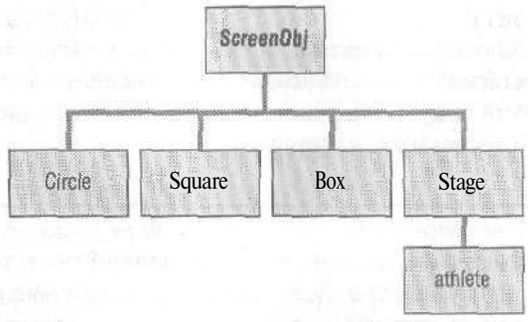


РИС. *17.3. Базовые и порожденные классы*

Базовый и производный классы в языке C++ иногда называют соответственно *родительским* (parent class) и *дочерним классом* (child class) или *классом предком* (ancestor class) и *классом потомком* (descendent class).

Производный класс сохраняет все характеристики базового класса. Это принято называть *наследованием* (inheritance). Кроме того, в производном классе имеются дополнительные данные и/или дополнительные функции.

Любой объект производного класса можно рассматривать, как объект базового. Так, любой объект класса Circle является также объектом класса ScreenObj, поскольку вы можете использовать его везде, где используются объекты класса ScreenObj, но не наоборот.



ПРИМЕЧАНИЕ Хотя каждый *мужчина* — это *человек*, но не каждый *человек* — это *мужчина*.

Замена унаследованных функций

По сравнению с классом ScreenObj, класс Circle не имеет дополнительных данных-членов. Однако, возможно, вас удивило (см. рис. 17.2), что мы повторили функции-члены show() и erase() в каждом классе (базовом и производном). Почему окружности и информационные рамки не наследуют эти функции-члены из базового класса ScreenObj?

Наследуют, но разве процедуры рисования окружности и информационной рамки не отличаются друг от друга? А что рисует функция show() класса ScreenObj? Возможны ситуации, в которых унаследованная функция не подходит, тогда необходимо создать другую функцию для выполнения эквивалентного действия.

В нашем примере, хотя экранный объект имеет координаты, размер и цвет, он не имеет формы. Поэтому функция show() класса ScreenObj ничего не будет рисовать! Информационные рамки, окружности, квадраты обладают формой, но поскольку формы у них разные, разные и процедуры их рисования. Следовательно, в каждом из этих классов должна быть своя функция show().

Но что же происходит с унаследованной функцией show()? Она ведь наследуется, не так ли? Верно. Однако она автоматически определяет, что если вызывается функция show() для объекта класса Circle, то нужно использовать именно тот вариант функции, который рисует окружности. В этом и есть сущность *полиморфизма* (polymorphism), о котором мы поговорим позже.



ПРИМЕЧАНИЕ Несмотря на то что функция show() класса ScreenObj является фиктивной, она тем не менее должна существовать. Если прототип функции не будет включен в объявление класса, то компилятор не даст объектам данного класса пользоваться этой функцией-членом.

Класс athlete

Класс athlete является производным от класса Stage, который, в свою очередь, является производным от класса ScreenObj. Все объекты класса ScreenObj имеют набор экранных координат центра объекта — переменных x, y и z. Хотя экран обладает лишь двумя измерениями, но с помощью координаты z можно разместить один объект перед другим.

Ниже представлено объявление данных-членов класса ScreenObj:

```
protected:
   float x, y, z;                  // Экранные координаты объекта
   int colorbrush, colorpen;       // Внутренний цвет и контур
   float size, lenght;
```

Поскольку все эти данные-члены являются *защищенными,* они не только наследуются, но, кроме того, могут вызываться функциями-членами производных классов Stage и athlete.

Несколько производных классов

В следующем примере производными от класса athlete будут два класса:

- Класс runner будет содержать объекты, похожие на гимнастов, но обладающие способностью «бегать». Бегуны будут двигать своими ногами и руками, переходя при этом на соседнюю ячейку экрана.
- Класс skater тоже будет содержать объекты, обладающие способностью «бегать». Однако бегать они будут на коньках, поэтому конькобежцы смогут скользить на каждом шаге сразу через несколько соседних ячеек.

Наиболее утомительной задачей в этих построениях является имитация движений руками и ногами. Вы можете сделать это, поняв, как был построен гимнаст, но я сэкономил вам время, разработав функции stepleft() и stepright().

На рис. 17.4 показаны гимнасты, двигающие при движении руками и ногами. Анимационная последовательность позиций — готов, шаг левой, готов, шаг правой, — показанная на рис. 17.5, создаст иллюзию, что гимнасты действительно двигают своими руками и ногами.



Шаг правой Шаг левой

Рис, *17.4,* *Гимнаст, выполняющий упражнения*

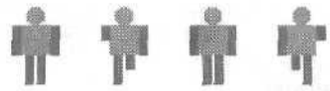


Рис. 17.5. *Анимационная последовательность для имитации бега*

Функция stepright() рисует гимнаста с поднятой правой ногой и левой рукой (бегущего от нас). Функция stepleft() делает то же самое с левой ногой и правой рукой. Обе функции могут получать аргумент в виде числа с плавающей точкой, задающей время в секундах, которое картинка будет оставаться в заданной позиции. Ниже приведены функции, взятые из программы c6run.h. Определение значений armsize и armwidth уже есть в заголовочном файле franca.h, и оно включено в первый комментарий просто для того, чтобы вы о нем не забывали:

```
// const int armsize=20, armwidth=6;
void runner::stepright(float time)     //c6run.h
{
   Clock any;
   erase();
   leftarm.resize(armsize/2, armwidth);
   leftarm.place((x-(armsize+armwidth)/2.), y-armsize/4);
   rightleg.resize(armsize/2, armwidth);
   rightleg.place(x+(armsize/2.+armwidth), y+3*armsize/4);
   show();
   any.wait(time);
}
void runner::stepleft(float time)
{
   Clock any;
   erase();
   rightarm.resize(armsize/2, armwidth);
   rightarm.place((x+(armsize+armwidth)/2.), y-armsize/4);
```

продолжение

```
    leftleg.resize(armsize/2, armwidth);
    leftleg.place(x-(armsize/2.+armwidth), y+3*armsize/4);
    show();
    any.wait(time);
}
```

Бегуны и конькобежцы

Зная, как рисовать гимнаста в этих новых позициях, можно догадаться, как примерно должна работать функция run():

Шаглевой

Готов

Сместиться на одну позицию

Шаг правой

Готов

Сместиться на одну позицию



СОВЕТ Помните, что удалять картинку нужно непосредственно перед тем, как ее двигать, поскольку функция erase() просто перекрашивает в белый цвет картинку, находящуюся в текущей позиции. Если вы сначала передвинете картинку, а затем ее удалите, исчезнет новая картинка, а старая останется.

Двигать бегуна очень просто. Поскольку бегун является гимнастом (и, соответственно, объектом класса Stage), то мы можем использовать функцию-член place(), чтобы разместить его в заданной позиции. Более того, поскольку нам известны координаты x и y, то можно легко вычислить его новую позицию.

В данной реализации бегун может двигаться по экрану только слева направо. Это означает, что на каждом шаге будет расти только координата x.

Ниже представлен один из возможных вариантов функции-члена run():

```
void runner::run()
{
    stepleft(.1);
    erase();
    ready(.1);
    stepright(.1);
    erase();
    ready(.1);
```

```
    eraseC);
    place(x+1, y);
    ready(0.);
}
```

Убедитесь, что вы сами нигде не изменяете значения экранных координат. Функция-член place() будет изменять координаты всех объектов, относящихся к объектам класса Stage, вычисляя разность между новыми координатами (заданными в качестве аргументов функции place()) и предыдущими координатами (значениями x и y).



ПРИМЕЧАНИЕ Если вы измените значение x или используете выражение x++ в качестве параметра, то новая и старая координаты будут совпадать и никакого движения не будет.

Объявление класса runner имеет следующий вид:

```
class runner:public athlete
{
protected:
   void stepleft(float);
   void stepright(float);
publ ic:
   void run();
};                    // Не забывайте ставить точку с запятой!
```

Полезно сделать функции-члены stepleft() и stepright() защищенными (protected), поскольку тогда доступ к ним из общих частей программы станет невозможным, но позволит им тем не менее оставаться функциями-членами производных классов (таких, как skater).

Объявление класса skater имеет следующий вид:

```
class skater:public runner
{
public:
   void run();
};
```

Подмена унаследованной функции

Обратите внимание, что функции stepleft() и stepright() в предыдущем примере повторно не объявляются. Они наследуются из базового класса runner. Что же касается функции run(), то может показаться странным, что мы объявляем функцию с тем

же именем, что была унаследована! Что же происходит в этом случае? Когда вы приказываете конькобежцу бежать, то он будет использовать функцию-член run(), объявленную в классе skater, а не ту, которая была унаследована. В этом и есть сущность *полиморфизма* (polymorphism). Вы можете определить несколько способов обработки заданного запроса (на вызов функции run()), в соответствии с тем, к какому классу относится данный объект.

Для конькобежца функция run() моделирует его скольжение по дистанции попеременно на каждой ноге. Вот одна из возможных реализаций функции:

```
void skater::run()
{
   for(int i=1; i<=5; i++)   // Перемещение на 5 шагов
   { // Остаемся на левой ноге
      stepleft{.1);
      erase();
      place(x+1, y);
   }
   ready(0.);
   for(i=1; i<=5; i++)      // Перемещение на следующие 5 шагов
   { // Остаемся на правой ноге
      stepright(.1);
      erase();
      place(x+1, y);
   }
   ready(0.);
}
```

Бежать обоих гимнастов можно заставить по следующей программе:

```
void mainprog()
{
   runner jane;
   jane.place(20, 80);
   skater julia;
   int k;
   for(k=1; k<=10; k++)
   jane.run();
}
```

А если вы захотите, чтобы конькобежец двигался как бегун? Он должен уметь это делать, поскольку конькобежец остается бегуном (так как класс skater является производным от класса runner). Для этого нужно вызвать ту функцию-член run(), которая объявлена в классе runner, а не в классе skater. Другими словами, нужно сообщить компилятору, что для объекта класса skater должна быть вызвана функция, объявленная в классе runner. Для этого используется уже известный нам меха-

низм уточнения имен — перед именем функции run() через два двоеточия указывается имя класса runner. Таким образом, чтобы объект julia класса skater двигался как объект класса runner, нужно написать следующие инструкции:

```
for(k=1; k<=10; k++)
julia.runner::run();
```

Определение полиморфизма

Полиморфизм по-гречески означает «много форм». Объекты, имеющие общего предка, могут принимать разные формы, оставаясь при этом схожими. Вспомните, бегун и конькобежец могут двигаться. Тем не менее конькобежец делает это иначе, чем бегун. Но вам не нужно постоянно заботиться об этом. Вы лишь сообщаете каждому из них, что нужно делать, и они побегут, кто как умеет.

Эта концепция уже использовалась с экранными объектами. Все они могли быть изображены, удалены и размещены в любом месте экрана. Тем не менее способ рисования квадрата отличался от способа рисования окружности, гимнаста, робота и т. д. Действительно, объекты класса ScreenObj имеют множество разных форм!

Класс Stage и полиморфизм

Объекты класса Stage представляют особый интерес, поскольку каждый из них может содержать в себе несколько экранных объектов, включая и объекты самого класса Stage. Например, вы можете включить в один объект гимнаста и бегуна (причем, атрибуты обоих наследуются от класса Stage). Когда объекту класса Stage предписывается появиться на экране, он в точности знает, как нужно изображать каждую свою часть. Тем не менее до сих пор мы в полной мере не использовали всех возможностей полиморфизма.

Допустим, например, что функция получает в качестве аргумента объект класса runner:

```
void march(runner volunteer)
{
   volunteer.run();
}
```

Если вы теперь посмотрите на следующую программу, то наверно решите, что объект smith будет двигаться как бегун, а объект yamaguchi — как конькобежец.

```
runner smith;
skater yamaguchi;
march(smith);
march(yamaguchi);
```

Но этого не произойдет. Функция march() умеет обращаться только с объектами класса runner, указанного в качестве ее параметра. Когда эта функция будет отком-

пилирована (переведена на машинный язык), то в ней не будет никакой информации об объектах других классов.

Только во время работы программы функция получит в качестве аргумента объект smith или yamaguchi. Но на данном этапе это поздно! Ведь функция march() уже переведена на машинный язык и содержит информацию только о том, как нужно использовать функцию run() для обработки объектов класса runner.

Чтобы повременить с выбором варианта функции run(), который надлежит использовать, можно определить ее как *виртуальную функцию* (virtual function). Для этого перед прототипом функции run() в объявлении класса runner ставится ключевое слово virtual:

```
virtual void run();
```

Эта инструкция указывает компилятору, что для функции run() не нужно назначать код из класса runner, а нужно отложить выбор конкретной версии функции до этапа выполнения программы, Естественно, в этом случае программа станет немного длиннее и скорость ее работы снизится.

Самостоятельная практика

- Модифицируйте функцию-член skater::run() так, чтобы перед началом скольжения, как это происходит на самом деле, конькобежец сделал несколько шагов (как бегун).
- Поскольку как конькобежец, так и бегун являются гимнастами, они должны уметь вести себя как гимнасты. Напишите программу, в которой объявляются объекты классов runner и skater, которые сначала должны выполнить подскок, потом пробежаться и затем снова выполнить подскок.
- Хотя бегун — это гимнаст, но гимнаст — это не бегун. Что произойдет, если вы, объявив гимнаста (объект класса athlete), прикажете ему бежать (вызовите для объекта функцию run())?
- Функция JumpJack() была разработана таким образом, чтобы в качестве аргументов получать объекты класса athlete. Тем не менее аргументами могут быть также объекты классов runner и skater. Попробуйте вызвать функцию с этими аргументами и посмотрите, что получится. C++ строго контролирует соответствие между типом параметров функции и типом ее аргументов (передаваемых при вызове функции). Почему тогда можно использовать функцию JumpJack() с объектами классов runner и skater, хотя она определена для использования с объектами класса athlete?
- Напишите функцию goaway(runner someone). Эта функция должна заставить объект класса runner, переданный ей в качестве аргумента, пробежать 10 шагов. Попытайтесь вызвать функцию не только с объектом класса runner, но и с объектом класса skater. Оцените результат.
- Включите в класс skater конструктор, который должен соединять ноги конькобежца с роликовой доской. Доску можно легко построить из прямоугольника и двух окружностей.



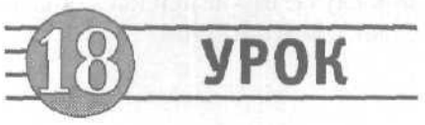
СОВЕТ Поскольку класс skater является производным от класса Stage, можно просто вставить (функция insert()) в объект класса skater прямоугольник и две окружности.

- Класс wagon в том виде, в котором он был создан, не позволяет пользователю размещать, удалять и отображать объекты этого класса на экране, поскольку класс wagon не является производным от класса ScreenObj. Создайте производную от класса Stage версию класса wagon. В этом случае вам не придется объявлять объект класса Stage — класс wagon уже будет таким объектом.

Что нового мы узнали?

В этом уроке мы научились

- Создавать производные классы из существующего базового класса.
- Наследовать данные и функции базового класса.
- В едином ключе работать с разными объектами одной иерархии классов.



Усложнение приложений

- Совершенствование торгового терминала
- Выбор программного и пользовательского интерфейса
- Выбор и реализация классов
- Имитация спутниковой системы

Умение создавать приложения — это суть профессии программиста. На постоянное развитие этого умения нацелена большая часть уроков этой книги. На уроке 18 мы продолжим оттачивать свое мастерство на примере двух коротких проектов:

- Усовершенствованного с использованием классов торгового терминала.
- Усовершенствованной с использованием классов модели планетной системы.

Совершенствование торгового терминала

Здесь мы узнаем, как трансформировать в объект торговый терминал, созданный на уроке 9. При этом мы постараемся использовать максимум атрибутов торгового терминала, о которых рассказывалось на уроке 9 (и которые были реализованы в программе c3sale2.cpp).

Принцип работы терминала останется прежним. Единственное, что нам сейчас нужно, — это понять, как трансформировать торговый терминал в объект. Это позволит нам на следующих уроках разработать интересные дополнения к проекту. А сейчас полезно вернуться к уроку 9 и еще раз выполнить программу c3sale2.cpp.

Выбор программного и пользовательского интерфейса

По мере вашего роста как программиста все больше людей начнут пользоваться созданными вами программами. Некоторые из них будут делать это в качестве конечных пользователей. Конечным пользователем мы будем называть пользователя, который понятия не имеет, как устроена программа. Например, это может быть кассир в магазине. Все, что он должен уметь, — это нажимать на нужные клавиши. Тем не менее необходимо понимать, что есть и другой тип пользователя.

Пользователь программист

В мире экстремально сложного программного обеспечения программисту часто приходится дорабатывать программы, созданные другими программистами. Вы уже использовали разработанные мной программы из заголовочного файла franca.h. Следовательно, вы являетесь пользователем моих программ со статусом программиста. Однако с помощью моих программ вы создаете собственные программы, у которых может быть свой конечный пользователь, совершенно не понимающий в программировании.

Когда вы разрабатываете программное обеспечение для пользователя программиста, то должны позаботиться о том, чтобы обслуживание этого программного обеспечения было максимально простым.

Наш текущий проект — это проект создания класса торговых терминалов. С помощью объектов этого класса программа будет генерировать конечные продукты — торговые терминалы.

Б нашем случае вы создаете класс, и вы же будете одним из его пользователей. Следовательно, вы станете пользователем собственного программного обеспечения. Однако так будет не всегда!

Программный интерфейс

Для реализации терминала в качестве класса имеется масса возможностей. Одна из них состоит в том, чтобы получить терминал, которые только создается и запускается. Это означает, что, кроме создания терминала, в основной программе потребуется вызвать только одну функцию.

Основная функция программы такого терминала может иметь следующий вид:

```
void mainprog()
{
  terminal cashregister;
  cashregister.operate();
}
```

Это действительно очень простой интерфейс для программиста, который будет пользователем вашего класса.

Такой класс, который все делает сам, возможно не лучший вариант. Вы можете рассмотреть возможность включения в класс некоторых операций, позволяющих программисту манипулировать объектом с большей гибкостью. Например, вы могли бы создать функции-члены, чтобы дать программисту возможность самому выводить на экран информационные рамки с данными или обслуживать за один раз только одного покупателя.

Интерфейс конечного пользователя

Интерфейс конечного пользователя останется тем, который был в программе c3sale2.cpp. Пока пользовательский интерфейс нравится клиенту, его лучше не трогать. При каждом изменении пользовательского интерфейса — перемещения информационной рамки из одного положения в другое, задание дополнительного вопроса и т. д. — пользователю придется затрачивать определенное время, чтобы привыкнуть к этим изменениям. Вот почему пользователям изменения иногда не нравятся!

Объявление и определение класса

Класс terminal должен сохранить большую часть переменных и объектов предыдущей программы. Следовательно, вариант объявления может быть следующим:

```
class terminal
{
protected:
   float tax, amount, price, saletotal, change;
   Box Price, Cur_Total, Saletotal, Amount, Change;
   void items();
public:
   terminal();
   void operate();
};
```

Обратите внимание, что в программе c3sale2.cpp мы инициализировали информационные рамки. В объявлении класса этого делать нельзя. Задачу инициализации рамок оставим конструктору.



ПРИМЕЧАНИЕ В объявлении класса нельзя устанавливать начальные значения.

Конструктор

Код конструктора относительно прост. Он выглядит слишком длинным только потому, что информационные рамки нужно размещать на экране и снабжать метками.

```
terminal::terminal()
{
   tax=ask("Enter the sale tax %")/100.;
   Prise.place(450, 100);
   Prise.label("Prise:");
   Cur_Total.place(450, 140);
   Cur_Total.label("Subtotal:");
   Saletotal.place(450, 180);
   Saletotal.label("Total Sale:");
   Amount.place(450, 220);
   Amount.label("Amount Tendered:");
   Change.place(450, 260);
   Change.label("Change:");
}
```

На рис. 18.1 показаны информационные рамки после совершения сделки.

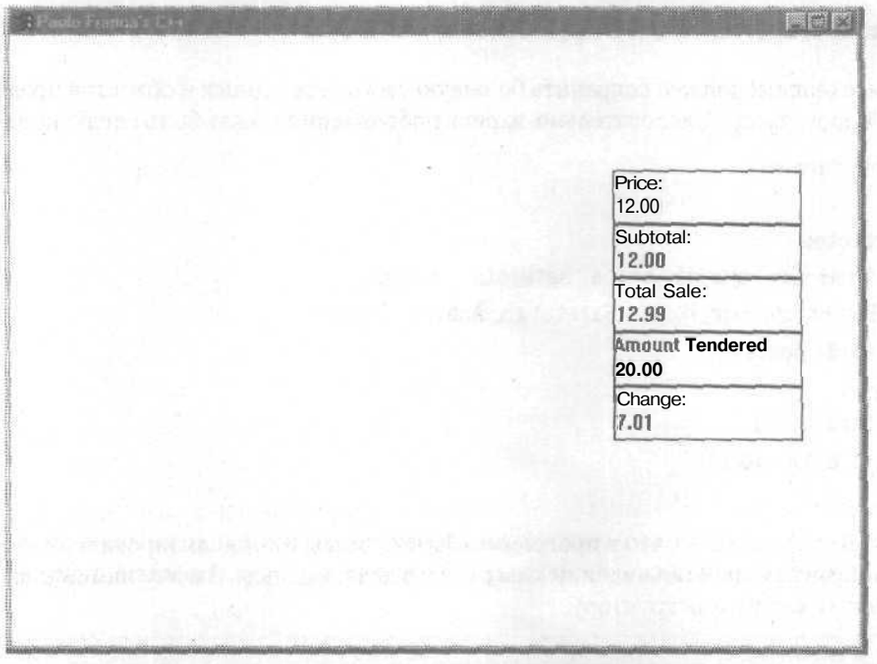


Рис. 18.1, Информационные рамки после совершения сделки

Функция-член operateQ

Функция-член operateQ — это именно та функция, которая заставляет терминал реально работать. Вы можете отметить, как эта функция почти в точности воспроизводит все то, что мы делали в предыдущей программе. Цикл управляет обслуживанием каждого покупателя. После каждой покупки на экран выводится полная стоимость и вычисляется сдача.

Чтобы упростить код, цикл, обрабатывающий покупку каждого товара, в свою очередь, обрабатывается еще одной функцией items(). Все по-прежнему точно повторяет программу c3sale2.cpp. Но в то же время обратите внимание, что пользователю программисту не понадобится вызывать функцию-член items(). Эта функция предназначена только для внутреннего использования. Именно по этой причине она закрыта.

Код функции operate() представлен ниже;

```
void terminal::operate()
{
   float tendered, change;
   for(;;)
   {
      saletotal=0;
      Price.say(" ");
```

```
      Cur_Total.say(" ");
      Saletotal.say(" ");
      Amount.say(" ");
      Change.say(" ");
      items();
      Saletotal.say(saletotal);
      amount=ask("Enter amount tendered:");
      Amount.say(amount);
      change=amount-saletotal;
      Change.say(change);
      if(yesno("Another customer?")) break;
   }
}
```

Отметьте, что все значения в информационных рамках перед началом очередной сделки стираются. Потом все товары обрабатываются функцией-членом items(), после чего все рамки снова заполняются соответствующей информацией.



ВНИМАНИЕ В функции operate() двумя очень схожими именами обозначены совершенно разные вещи: объект Change (с прописной буквы) типа Box и переменная change (со строчной буквы).

Функция-член items()

Функция items(), член класса terminal, играет ту же роль, что и функция getitems() в программе c3sale2.cpp. Она складывает стоимость всех товаров, приобретенных данным покупателем.

Код функции items() имеет следующий вид:

```
void terminal::items()
{
   for(;;)
   {
      price=ask("enter the price:");
      saletotal=saletotal+price;
      Price.say(price);
      Cur_Total.say(saletotal) ;
      if(yesno("Another item?")) break;
   }
   saletotal=saletotal*(1+tax);
   Saletotal.say(saletotal);
}
```



ПРИМЕЧАНИЕ Наиболее существенным улучшением программы торгового терминала была бы замена ввода цены на считывание штрих-кода. Именно этот принцип воплощен в большинстве сегодняшних торговых терминалов. Сейчас вы еще не знаете, как это реализовать, но узнаете на следующих уроках.

Программный код проекта

Полный программный код данного проекта можно найти на прилагаемой дискете в файлах c6term.cpp и c6term.h.

Основная программа c6term.cpp

Ниже представлен код программы c6term.cpp:

```
tfinclude "franca.h"
#include "c6term.h"
void mainprog()
{
   terminal cashregister;
   cashregister.operate();
}
```

Заголовочный файл c6term.h

Ниже представлен код заголовочного файла c6term.h:

```
ffifndef   C6TERM_H
ffdef ine C6TERM_H
#include "franca.h"
class terminal
{
protected:
   float tax, amount, price, saletotal, change;
   Box Price, Cur_Total, Saletotal, Amount, Change;
   void items();
public:
   terminal();
   void operate();
};
terminal::terminal()
{
   tax=ask("Enter the sale tax %")/100.;
   Prise.place(450, 100);
```

```
    Prise.label("Prise:");
    Cur_Total.place(450, 140);
    Cur_Total.label("Subtotal:");
    Saletotal.place(450, 180);
    Saletotal.label("Total Sale:");
    Amount.place(450, 220);
    Amount.label("Amount Tendered:");
    Change.place(450, 260);
    Change.label("Change:");
}
void terminal:: itemsO
{
    for(;;)
    {
        price=ask("enter the price:");
        saletotal=saletotal+price;
        Price.say(price);
        Cur_Total.say(saletotal);
        if(!yesno("Another item?")) break;
    }
    saletotal=saletotal*(1+tax);
    Saletotal.say(saletotal);
}
void terminal::operate()
{
    floattendered,change;
    for(;;)
    {
        saletotal=0;
        Price.say(" ");
        Cur_Total.say(" ");
        Saletotal.say(" ");
        Amount.say(" ");
        Change.say(" ");
        items();
        Saletotal.say(saletotal);
        amount=ask("Enter amount tendered:");
        Amount.say(amount);
```

продолжение

```
        change=amount-saletotal;
        Change.say(change);
        if(yesno("Another customer?")) break;
    }
}
#end if
```

Проект имитации нескольких спутников

Программа c5stars.cpp, разработанная *в* качестве проекта урока *15,* будет сейчас переработана с использованием объектов. В самом деле, мы представляли звезду (Солнце), планету (Землю) и спутник (Луну) *с* помощью окружностей.

Вообще говоря, эти объекты вращаются вокруг друг друга на определенном расстоянии и с определенными угловыми скоростями. При абстрактном подходе можно не делать различий между звездами, планетами и спутниками, а считать спутниками все эти объекты.

В самом деле, Земля, равно как и все остальные планеты солнечной системы, представляет собой спутник, вращающийся вокруг Солнца. Само Солнце тоже можно представлять себе как спутник, вращающийся вокруг самого себя с нулевой скоростью. Такой абстрактный подход приводит к идее создания нового класса объектов — спутников.

С помощью этого класса можно упростить программу c5stars.cpp, а также использовать ее в других приложениях.

Спутник можно представить в виде окружности, двигающейся по круговой орбите вокруг заданной точки, или, что более интересно, вокруг другого спутника (возможно, даже самого себя).

Помимо всех обычных атрибутов окружности, у спутников есть еще следующие:

- Планета, вокруг которой они вращаются.
- Расстояние до центра этой планеты — радиус орбиты.
- Угловая скорость.
- Текущий наклон орбиты наряду с радиусом задающий полярные координаты спутника относительно планеты.

К тому же, спутники должны уметь перемещаться с течением времени из одной точки в другую. Для вычисления текущего положения спутника можно определить функцию move(). Она будет вызываться для отображения на экране очередного кадра (тридцать раз в секунду) и одновременно рассчитывать для этого кадра новые координаты спутника.

Разработка класса satellite

Ниже представлено объявление класса satellite:

```
class satellite:public Circle
{
protected:
   float xplanet, yplanet;
   float radius, omega, wspeed;
public:
   satellite();
   move();
};
```

Данными-членами являются: координаты планеты (xplanet и yplanet), радиус орбиты (radius), текущий наклон орбиты (omega) и угловая скорость (wspeed).

Имеется несколько способов определения угловой скорости. В конечном счете вам нужно знать, на сколько радиан сдвигается спутник за время кадра (одну тридцатую секунды). Однако лучше, чтобы пользователь вашего класса (и вы тоже) мог задавать угловую скорость в градусах на секунду, даже если вы потом преобразуете это значение и сохраняете его в радианах на длительность кадра.

Конструктор

Включать в класс конструктор всегда полезно. Благодаря конструктору мы можем задать цвет, размер, начальное положение, а также решить другие задачи инициализации объекта. Чтобы перемещать спутник в новое орбитальное положение по истечении времени каждого кадра, воспользуемся знакомой нам по предыдущим урокам функцией move().

Очевидно, что эта функция могла бы рассчитывать текущее время и положение.

К несчастью, приведенное выше объявление класса нам пока не подходит. Как при таком объявлении задавать значения данных-членов?

Один из возможных вариантов решения проблемы состоит в том, чтобы позволить конструктору получать эти значения в качестве аргументов. Такой вариант мог бы работать, однако, поскольку конструктор предназначен исключительно для инициализации, изменить эти значения на новые без создания нового объекта уже не удастся. Следовательно, остаются две альтернативы:

- Открытые данные-члены.
- Функция-член для изменения значений данных-членов.

Некоторые данные-члены (например, радиус орбиты, координаты планеты или угловая скорость) остаются постоянными. Но даже с оставшимися значениями можно многого добиться, если проявить немного гибкости.

Функции-члены могут быть включены в класс для изменения следующих значений:

- Координат планеты.
- Радиуса орбиты.
- Угловой скорости.

Задание координат спутников

Как вы могли заметить, в программе c5stars.cpp очень удобно переходить из полярных координат в декартовы. Включите для этой цели в класс функцию polarxy(), и наш класс наконец будет готов. Вот соответствующее объявление:

```
class satellite:public Circle
{
protected:
   float xplanet, yplanet;
   float radius, omega, wspeed;
   void polarxy(float r, float theta,
            float x0, float y0, float &x, float &y);
public:
   satellite();
   void center(float x, floaty);
   void dist(float rad);
   void angle(float ang);
   void speed(float ws);
   void move();
   void center(satellite &another);
};
```

Для смены центров вращения в классе имеются две функции center(). С их помощью можно задать вращение планеты вокруг заданной точки *(x,y)* или вокруг другой планеты.

В этом случае можно было модифицировать функцию polarxyQ, создав ее версию без аргументов. Возможно, это и неплохая идея, но мы пока оставим ее, чтобы можно было использовать код нашей прежней функции.

Реализация класса

Теперь, когда объявление класса готово, можно разработать код его функций.

Конструктор

Ниже представлен программный код конструктора:

```
satellite::satellite()
{
   color(2);
   xplanet=yplanet=0;
   radius=0;
   omega=wspeed=0;
}
```

В этом конструкторе нет ничего особенного. Отметим только, что цвет спутника можно изменить в любой точке, поскольку функция color() в классе Circle открыта.

Смена центров вращения

Следующие функции изменяют координаты планеты:

```
void satellite::center(float xc, floatyc);
{
   xplanet=xc;
   yplanet=yc;
}
void satellite::center(satellite &another);
{
   center(another.x, another.y);
}
```

Заметили что-нибудь интересное в этом коде? Почему вторая функция использует координаты x и y другой планеты? Разве эти координаты не защищены?

Разумеется, да! Однако защищенные члены могут использоваться функциями-членами того же или производного класса. Поскольку вторая функция представляет собой функцию-член класса satellite, который является производным от класса Circle, то доступ к координатам становится возможным. Доступны не только координаты этого спутника, но и координаты любого другого спутника, который упоминается внутри функции-члена.

Для работы с объектами класса satellite гораздо удобней вторая версия функции center(), поскольку вместо определения координат Земли, чтобы включить их в вызов функции, можно просто написать следующую инструкцию:

```
Moon.center(Earth);
```

Изменение значений данных-членов

Функции для задания радиуса орбиты, текущего наклона орбиты и угловой скорости имеют следующий вид:

```
void satel lite: :dist(float rad)
{
   radius=rad;
}
void satellite::angle(float ang)
{
   omega=ang;
f
void satellite; :speed(fioat ws)
{
   wspeed=ws;
}
```

Движение по орбите

Наконец, функция для перемещения спутника из одной позиции в другую выглядит следующим образом:

```
void satellite::move()
{
   omega=omega+wspeed;
   polarxy(radius, omega, xplanet, yplanet, x, y);
}
```

Просто, не правда ли? Нужно только увеличивать наклон орбиты и затем рассчитывать новые декартовы координаты. Больше нет необходимости вызывать функцию place(x,y). Вызов функции move() обновляет значения декартовых координат напрямую.

Новая программа проекта планетной системы

Когда доступен класс satellite (заголовочный файл c6satell.h), то можно написать новую версию программы c5stars.cpp — программу c6sat.cpp,

Объявить и инициализировать объекты можно следующим образом:

```
const float pi2=2*3. 14159;
Stage universe;
Clock timer, sidereal;
satellite Sun, Earth, Moon;
Sun. resize(40);
```

```
Sun.origin(320, 240);
Sun.scale(1. -1.);
Earth.dist(120);
Earth.speed(pi2/36.5/30.);
Earth.center(Sun);
Moon.resize(12);
Moon.dist(80);
Moon.color(3);
Earth.color(4);
Moon.speed(pi2/2.8/30.);
universe.insert(Earth);
universe.insert(Moon);
Sun.show();
```

Анимационный цикл достаточно прост:

```
for(; sidereal.time()<36.5;)
{
   Earth.move();
   Moon.center(Earth);
   Moon.move();
   universe.show();
   timer.watch(.033);
   timer.reset();
   universe.erase();
}
```

Почему Луна должна вращаться вокруг Земли непосредственно внутри цикла? Нельзя ли это сделать один раз до начала цикла так, как это сделано для Солнца? Нет, нельзя!

Проблема в том, что координаты Земли меняются, и Луне каждый раз приходится передавать их новые значения. Для неподвижного Солнца необходимость в такой процедуре отсутствует. Теперь вы видите, почему полезно иметь возможность смены координат центра вращения?

Внутренний космос

Для чего еще могут пригодиться спутники? Электроны в атоме также являются спутниками ядра.

Можно написать программу имитации движений электронов в атоме так же легко, как вы написали предыдущую программу. Если вам нужно, чтобы вокруг ядра вращалось 10 электронов, просто объявите 10 спутников, инициализируйте и двигайте их! К сожалению, картина пока не объемна. Но потерпите немного. Скоро вы познакомитесь с массивами!

Что нового мы узнали?

В этом уроке мы научились

- Выбирать подходящие классы для создания приложений.
- Совершенствовать свои прежние разработки.

Часть VII

Массивы и структуры

Массивы очень удобны при обработке нескольких однотипных объектов, поскольку исключают необходимость присваивать каждому объекту собственное имя. Это значительно упрощает программы, так как код не зависит от обрабатываемого объекта.

В части VII вы научитесь работать с массивами объектов любого класса, в том числе и с числовыми массивами. Как правило, слова, фразы и текст можно также обрабатывать с помощью символьных массивов. В конце части вы продолжите развитие своих навыков создания приложений, усовершенствовав проекты торгового терминала и спутниковой системы.

19 УРОК Массивы

- Объявление и использование массивов
- Массивы массивов
- Числовые массивы
- Сортировка массивов

На этом уроке, чтобы вы поняли назначение массивов и могли оценить их преимущества для идентификации групп однотипных объектов, мы воспользуемся нашими знакомыми гимнастами. После того как вы с помощью массивов гимнастов усвоите основные концепции, мы перейдем к наиболее распространенному виду массивов — числовому массиву.

Вы научитесь объявлять массивы, получать доступ к элементам массивов, использовать их в выражениях, передавать в качестве аргументов и получать в качестве параметров функций.

Работа с массивами

Массив (array) представляет собой группу объектов одного класса, причем вы можете обратиться к любому объекту по его положению в строке. Например, если гимнасты построены в шеренгу, но вы не знаете их имена, проще всего обращаться к ним в соответствии с их положением в шеренге: первый, второй, третий...

Важно, чтобы вы понимали достоинства массивов. Возьмем простейший случай, допустим, что вам требуется написать программу, заставляющую пятерых гимнастов, показанных на рис. 19.1, выполнять упражнения.

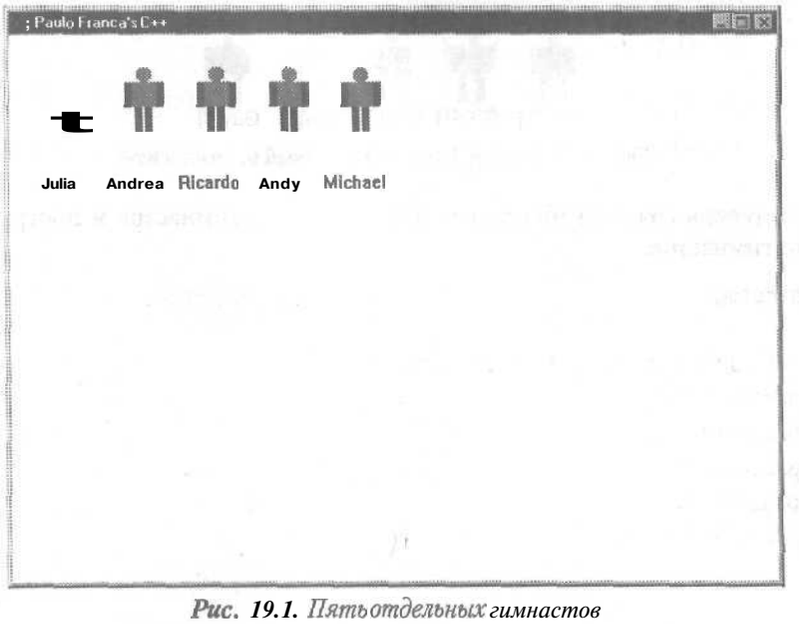


Рис. 19.1. *Пять отдельных гимнастов*

Пусть вы создали эту пятерку и с помощью следующей инструкции присвоили им имена:

```
athlete Julia, Andrea, Ricardo, Andy, Michael;
```

Затем вы вынуждены каждому из них давать команду на выполнение одного и того же упражнения:

```
JumpJack(Julia);
JumpJack(Andrea);
JumpJack(Ricardo);
JumpJack(Andy);
JumpJack(Michael);
```

Это, конечно, не очень сложно, но как быть, если гимнастов сотня? А если их тысяча?

Стандартный метод решения этой проблемы состоит в том, чтобы дать имя всей группе гимнастов, а затем обращаться к каждому конкретному гимнасту как к первому, второму, третьему...

Выглядеть это будет примерно так:

Обозначить именем Guy группу из пяти гимнастов

Повторять значение *индекса* (index) от 1 до 5:

Элементу Guy в позиции *индекс* выполнить упражнение

Как мы знаем по изучению циклов, компьютер поймет, что все члены группы должны выполнить упражнение, но теперь вам нет необходимости писать тысячи строк кода, независимо от того, сколько элементов массива Guy вы хотите использовать! На рис. 19.2 показан массив гимнастов.



Рис. 19.2. *Массив Guy, построенный из гимнастов*

Давайте теперь сравним программу для отдельных гимнастов и программу для массива гимнастов:

```
void mainprog()
{
   athlete Julia, Andrea, Ricardo, Andy, Michael;
   JumpJack(Julia);
   JumpJack(Andrea);
   JumpJack(Ricardo);
   JumpJack(Andy);
   JumpJack(Michael);
}
```

```
void mainprog()
{
   athlete Guy[5];                    // Объявление массива из пяти гимнастов
   for (int which=0; which<=4; which++);
   {
      JumpJack(Guy[which]);           // Выполнение упражнения каждым гимнастом
   }
}
```

На последнюю программу в отличие от первой практически не влияет количество гимнастов (меняется только Значение переменной в условии завершения цикла).

Вы не забыли, что массив составляется из объектов одного и того же класса? Таким образом, при желании ту же программу можно использовать для бегунов:

```
void mainprogO
{
   runner Guy[5];                     // Объявление массива из пяти бегунов
   for (int whicn=0; which<=4; which++);
   {
      Guy[which].run();               // Перемещение каждого бегуна
   }
}
```

Варианты использования массивов

Вы можете использовать массивы объектов любых типов: гимнастов, бегунов, роботов, окружностей и даже самых простых, например элементами массива могут быть переменные типа int или float. На языке C++ элементы массива помечаются целыми значениями (int) индекса. Первому элементу массива присваивается значение индекса, равное 0, второму — 1 и т. д.

Б предыдущей программе массив Guy содержал пять элементов:

Элемент Guy[0] был первым

Элемент Guy[1] был вторым

Элемент Guy[2] был третьим

Элемент Guy[3] был четвертым

Элемент Guy[4] был пятым

Элемент Guy[5] не существовал, так как элементов всего пять!



ВНИМАНИЕ Это традиционный программный жучок: объявить массив из N элементов и затем пытаться использовать элемент с номером N, которого, разумеется, не существует. Индексы элементов массива начинаются с 0 и заканчиваются индексом N–1. Не забывайте о существовании нулевого элемента.

Выражения в качестве индекса массива

Объявление массива похоже на объявление любого другого объекта за исключением того, что после имени массива в квадратных скобках указывается число его элементов. Например, рассмотрим следующую инструкцию:

```
int points[10];
```

В этой инструкции объявляется массив с именем points из 10 целых элементов (пронумерованных от 0 до 9).

В следующей инструкции объявляется массив measurements из 12 элементов с плавающей точкой (пронумерованных от 0 до 11):

```
float measurements[12];
```

Имя массива обозначает всю группу элементов. Имя массива, за которым в квадратных скобках следует значение индекса, обозначает конкретный элемент. Например, для первого примера этого раздела следующий идентификатор обозначает группу из 10 элементов:

```
points
```

А для обращения к конкретному элементу группы (второму) используется такое обозначение:

```
points[1]
```

До тех пор пока значение выражения остается целым, его можно указывать в качестве индекса:

- Обозначение Guy[which] указывает на конкретный элемент массива Guy. Если вы знаете значение выражения which, то легко можете определить этот элемент.
- Обозначение Guy[which-1] указывает на другой конкретный элемент массива Guy.

Однако значение индекса не может указывать на несуществующий элемент массива. В обозначении Guy[which] диапазон значений выражения which должен оставаться между 0 и 4 включительно. Любое другое значение выходит за границы массива. Контролировать нарушение границ массивов в C++ должен программист[1].

Неправильное значение индекса

Основная причина ошибок в программах состоит в присваивании индексу неправильного значения. Использовать выражения в качестве индекса приходится довольно часто. При этом вы можете не знать наверняка, какие значения будет принимать это выражение при выполнении программы. Например, пусть вы объявили массив List из 10 элементов (пронумерованных от 0 до 9). Допустим, далее в программе вы используете следующую инструкцию:

---

[1] В отличие от многих других языков программирования, в которых эта задача возложена на компилятор. — *Примеч. ред.*

```
list[(k-3)*j]=0;
```

Значения переменных k и j определяют значение индекса (которое должно быть в интервале от 0 до 9 включительно). Вполне вероятно, что в своей программе вы не сможете проконтролировать все факторы, влияющие на значения переменных k и j. Таким образом, результатом выполнения выражения в квадратных скобках может оказаться, например, значение -13.

Поскольку элементы массива располагаются в памяти компьютера последовательно, компилятор обязательно найдет то место, где должен находиться элемент массива List[-13], и поместит туда ноль, хотя это место в памяти вполне вероятно, уже используется другим фрагментом программы. Последствия будут непредсказуемы!

Массивы в выражениях

В выражениях обычно используются числовые массивы, элементами которых являются числа, например значения типа int или float. Поскольку каждый элемент такого массива является числом, его значение и подставляется в выражение.

Рассмотрим следующий фрагмент программы:

```
void mainprog()(
{
   int value[10];                  // Объявление массива из десяти целых
   for (int index=0; index<=9; index++)
   {
      value[index]=9-index;        // Вычисление значения каждого элемента
   }
...
```

Можете ли вы определить значение каждого элемента массива после выполнения этого фрагмента программы? Каково будет значение элемента с индексом О? А значение элемента с индексом 1?

Обратите внимание, что в цикле мы вычисляем выражение 9-index и сохраняем результат в элементе массива value[index]. Следовательно, после выполнения этого фрагмента массив заполнится числовыми значениями, которые затем можно будет тем или иным способом использовать.

Можно, например, отображать значение каждого элемента массива в информационной рамке:

```
voidmainprog(}(
{
   int value[10];              // Объявление массива из десяти целых
   for (int index=0; index<=9; index++)
```

продолжение

```
  {
    value[index]=9-index; // Вычисление значения каждого элемента
  }
  Box Sal;
  for (which=0; which<=9; which++)
  {
    Sal.say(value[which]);
  }
}
```

Вы заметили, что мы использовали элемент массива в качестве аргумента функции? Я надеюсь, вы самостоятельно проверите правильность ваших предположений по поводу значений элементов массива.

Массивы в качестве аргументов

Массивы и их элементы могут также быть аргументами функций. Важно, чтобы вы точно представляли себе, какой тип объекта вы используете. Например, рассмотрим следующее объявление:

```
athlete Guy[10];
```

С учетом этого объявления следующие две инструкции означают нечто совершенно разное:

```
JumpJack(Guy);
JumpJack(Guy[1]);
```

Первая инструкция вообще ошибочна! Можете понять, почему? У функции JumpJack() всего один параметр типа athlete, но Guy — это массив гимнастов (массив объектов типа athlete), а одного гимнаста обозначает аргумент Guy[l]l

Аргументом функции JumpJack() может быть каждый гимнаст массива, но не целый массив гимнастов, поскольку единовременно эта версия функции в состоянии обрабатывать только один объект. Тем не менее мы можем определить еще одну версию функции JumpJack() с массивом гимнастов в качестве параметра. В этом случае следующая инструкция будет правильной:

```
JumpJack(Guy);
```

Массивы в качестве параметров

Когда параметром функции является массив, то заголовок функции должен отражать этот факт. Например:

```
void jumpjack(athlete eachone[10], int howmany_athletes)
{
   for(int i=0; i<howmany_athletes; i++);
   {
      eachone[i].ready();
      eachone[i].up();
      eachone[i].ready(0.);
   }
}
```

Эту версию функции JumpJack() можно вызывать с массивом гимнастов в качестве аргумента. Если вы уверены, что гимнастов всегда будет 10, можете опустить аргумент howmany_athletes.

Б данном случае в заголовке функции четко обозначено, что eachone является массивом. Это указывает компилятору, что индексы должны быть с ним связаны. Другим интересным фактом является то, что фактически вам не нужно задавать размер массива. Так как последний уже объявлен (где-то в другом месте программы), то повторно (в данном случае в функции) область памяти ему выделена не будет, поскольку в C++ по умолчанию массивы передаются по ссылке. Функции надо указать только то, что eachone — это массив и, значит, может сопровождаться индексом.

Таким образом, заголовок функции может быть следующим:

```
void jumpjack(athlete eachone[], int howmany_athletes)
```



ПРИМЕЧАНИЕ Если у вас имеется многомерный массив (вы узнаете о таких массивах *позднее*), то в заголовке функции только последняя скобка может оставаться пустой.

Программа c7jack.cpp

Использование массива на примере массива гимнастов, выполняющих упражнение, показано в программе c7jack.cpp.

```
//                c7jack.cpp
// Иллюстрация использования массивов
//                31.07.1994
//
#include "franca.h";
```

продолжение

```
athlete Guy[7];
void JmpJack(athlete somebody)
{
   somebody.up();
   somebody.ready();
}
void mainprog()
{
   for (int i=0; i<7; i++)
   {
      JumpJack(Guy[i]);
      Guy[i].say("Done!");
   }
}
```

Эта программа почти не зависит от того, сколько будет гимнастов: один или несколько. Отличия будут очень несущественны:

- Необходимое число элементов задается при объявлении массива.
- Цикл должен начинаться с *нуля* до значения последнего элемента массива.

Число элементов массива

Если вы объявляете массив из семи элементов, но используете только три из них, программа будет исправно работать. Единственная проблема в том, что вы непродуктивно занимаете память компьютера. С другой стороны, обращение к элементу с большим, чем было объявлено, индексом может вызвать ошибку.

Часто бывают ситуации, в которых размер массива заранее определить невозможно. Например, вы изучаете возраст студентов в классе. Если в классе 28 студентов, то, возможно, вы решите создать программу, рассчитанную именно на это количество. Однако в другом классе ваша программа окажется неработоспособной.

Одно из решений Состоит в том, чтобы объявить массив следующим образом:

```
int student_age[28];
```

Теперь можно изменить значение 28 на любое другое, скажем на 32, для большего класса. Но это не слишком удачное решение, поскольку в другом месте программы тоже может потребоваться число студентов. Например, если вы вычисляете средний возраст студентов, то у вас наверняка имеется следующий фрагмент программы:

```
sum=0;
for(int i=0; i<28; i++);
{
      sum=sum+student_age[i];
}
average=sum/28;
```

Обратили внимание, что уже в двух местах программы 28 нужно поменять на 32? В данном примере это достаточно просто, но если у вас длинная программа, найти в ней все такие места будет уже гораздо сложнее!

Новичок, вероятно, все равно решит каждый раз переделывать свою программу для разных классов с разным числом студентов. Это далеко не лучший способ!



СОВЕТ Не забывайте, что вы, как профессиональный программист, не должны адаптировать свою программу к каждому запуску.

Адекватное решение проблемы состоит в хранении числа студентов в виде отдельной переменной с собственным именем.

```
int number_of_students;
```

Значение этой переменной можно считывать с клавиатуры или передавать в качестве аргумента функции. Массив в этом случае мог бы объявляться с достаточным, чтобы предусмотреть все возможные варианты, числом элементов. Например:

```
int student_age[100];
```

Теперь фрагмент программы расчета среднего возраста студентов будет выглядеть так:

```
sum=0;
for (int i=0; i<number_of_students; i++);
   {
      sum=sum+student_age[i];
   }
average=sum/number_of_students;
```

Задание **размеров** массива с помощью **константы**

При объявлении массива для обозначения его размера нельзя использовать переменную. Это может быть только константа, причем целая. Например, если n — это не константа, то следующее объявление неправильно:

```
athlete Guy[n];
```

Значение переменной становится известным *только* в процессе выполнения программы. Кроме того, в процессе выполнения программы переменная, как следует из ее названия, может менять свое значение. Но перед тем как начнется выполнение программы, компилятор должен выделить область памяти для хранения всех переменных (включая элементы массива). Чтобы компилятор мог зарезервировать места для всех элементов массива, их число должно быть известно заранее. Когда при запуске программы переменным начинают присваиваться их значения, вся область памяти уже должна быть распределена. Именно по этой причине размер массива задается с помощью константы. Например, для задания размера массива можно объявить именованную константу:

```
const int arraysize;
float prices[arraysize];
```

При этом не забывайте, поскольку arraysize является константой, ее значение в программе изменить нельзя.

Самостоятельная практика

О Измените программу c7jack.cpp так, чтобы можно было задавать число гимнастов. Пусть каждый из них делает следующее:

- Говорит «Hi».
- Выполняет подскок.
- Называет свой номер (0, 1, 2...).

Числовые массивы

Во многих ситуациях необходимы массивы, элементами которых являются числовые значения. Это могут быть массивы типа int, Long, float или double.

Массив массивов

Элементами массивов могут быть объекты разных типов. Но возникает вопрос: бывают ли массивы, элементы которых тоже являются массивами?

Допустим, что одна группа гимнастов приходит на занятия в восемь часов. Если представить группу массивом, то гимнастам нужно присвоить индексы — О, 1, 2... Теперь предположим, что другая группа гимнастов приходит в девять. Очевидно, что эту группу тоже можно рассматривать в качестве массива.

Пусть у нас есть три такие группы гимнастов, каждая из которых является массивом:

О Группа 0 (восьмичасовая).

О Группа 1 (девятичасовая).

О Группа 2 (десятичасовая).

Но теперь можно заметить: то, что мы называем *группой,* фактически тоже является элементом массива. Пусть в каждой группе этого нового массива занимается, например, по четыре гимнаста. Этот вариант показан на рис. 19.3.

Мы можем объявить такой двумерный массив следующим образом:

```
athlete Guy[3][4];
```

Таким образом, объект Guy представляет собой массив из трех элементов, каждый из которых в свою очередь является массивов из четырех гимнастов.



Рис 19.3. *Массив массивов*

Строки и столбцы массива

Если вы посмотрите на рис. 19.3, то заметите, что в каждой из трех строк стоит по четыре гимнаста. Кроме того, каждая строка состоит из гимнастов, входящих в одну группу. Таким образом, когда мы говорим о *Нулевой группе*, мы фактически говорим о *Нулевой строке.*

Когда элементы расположены в виде таблицы (как гимнасты на рис. 19.3), вы можете обозначать каждый элемент соответствующими значениями строки и столбца, на пересечении которых он расположен. Таким образом, элемент Guy[2][3] обозначает гимнаста на второй строке третьего столбца. На рис. 19.4 показаны строки и столбцы двумерного массива гимнастов.

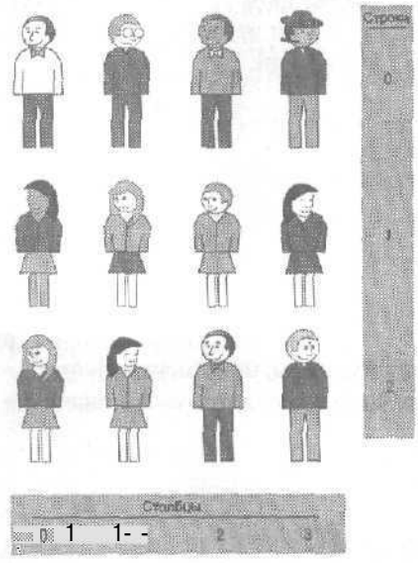


Рис. 19.4. *Строки и столбцы двумерного массива гимнастов*

Инициализация массива

Так же как вы отвечаете за каждый объект или переменную своей программы, вы, как программист, отвечаете за задание правильных значений элементов массивов. При создании массива его элементы содержат всякий «мусор», оставшийся в памяти компьютера от прежних программ. Не нужно полагать, что эти элементы имеют нулевые или любые другие известные значения.

Вы можете постепенно присвоить значения всем элементам массива в процессе выполнения программы или при объявлении массива присвоить все начальные значения сразу. Этот последний вариант особенно удобен для числовых массивов. В этом случае следом за объявлением массива ставится знак равенства и в фигурных скобках через запятую перечисляются все начальные значения. Первое значение соответствует первому элементу (с индексом ноль), второе значение — второму элементу (с индексом 1) и т. д.

```
int values[10]={9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

Приведенное выше объявление присваивает значение 9 элементу [0], значение 8 — элементу [1], значение 7 — элементу [2] и т. д.

Отметьте следующие элементы синтаксиса:

- Знак равенства после квадратных скобок.
- Запятые между начальными значениями отдельных элементов.
- Фигурные скобки для начальных значений всех элементов.

Рассмотрим следующую инструкцию:

```
int matrix[2][3]={{0, 0, 1}, {2, 3, 4}};
```

Если таким образом инициализировать двумерный массив, то первым меняется значение последнего индекса. При этом инициализация эквивалентна выполнению следующей группы инструкций:

```
matrix[0][0] = 0;
matrix[0][1] = 0;
matrix[0][2] = 1;
matrix[1][0] = 2;
matrix[1][1] = 3;
matrix[1][2] = 4;
```

Обратите внимание, что в этой последовательности инструкций сначала значение первого индекса остается нулевым, пока значение второго меняется от 0 до 2. Затем значение первого индекса изменяется на 1, а значение второго опять меняется от 0 до 2.

Использование массивов

Допустим, вы хотите хранить в массиве возраст пяти гимнастов группы. Вы можете объявить массив гимнастов и массив возрастов следующим образом:

```
athlete guy[5];
int age[5];
```

Пусть каждый гимнаст сообщает свой номер, запрашивает у вас свой возраст и затем сообщает его.

Программа c7age.cpp

Описанная выше процедура реализована в программе c7age.cpp.

```
//                  c7age.cpp
// Иллюстрация использования целочисленных массивов
//                  31.07.1994
//
tfinclude "franca.h"
athlete Guy[5];
void mainprog()
{
   int age[5];
   for (int i=0; i<=4; i++)
   {
      Guy[i].ready();
      Guy[i].say(i);
      age[i]=Guy[i].ask("What is my age?");
   }
   for (i=0; i<=4; i++)
   {
      Guy[i].say(age[i]);
   }
}
```

Числовые массивы

Как вы могли заметить по предыдущему примеру, мы не слишком часто пользовались массивом athletes. Основная работа выполнялась с массивом age. Действительно, в реальной жизни часто приходится иметь дело с простыми числовыми

массивами. Это может быть массив возрастов студентов в классе или массив балансов в счетах покупателей. Программа не будет сильно отличаться, если вместо обоих массивов у нас останется только массив age.

Самостоятельная практика

- Напишите еще одну программу только с массивом age. Программа должна определять положение в массиве самого старшего студента. Для ввода значений используйте функцию ask(), а для вывода результатов информационную рамку.

Нежелательные массивы

Массивы нужны только тогда, когда требуется держать вместе все значения, которые могут понадобиться в будущем. Если нет других веских причин использовать массивы, то лучше еще раз оценить их реальную необходимость. Например, если вы хотите ввести несколько значений и вычислить их среднее значение, то не обязательно это делать с помощью массива. Можно складывать значения по мере считывания, а затем разделить результат на число считанных значений:

```
Box result("Average:");
int n;                    // Число элементов
float value, total;
n=ask("Input number of elements");
total = 0;
for(int i=1; i<=n; i++);
{
   total=total+ask("Input a number");
}
total = total/n;
result.say(total);
```

В предыдущем примере числа накапливались по мере их ввода. При этом необходимость в использовании массивов для хранения чисел не возникала. Вообще же массивы могут непроизводительно потреблять слишком большие объемы памяти и при этом делают программу менее читабельной.

Создание нового типа

Пусть у вас имеется несколько одинаковых объявлений массивов, например:

```
float matrix[5][3], prices[5][3], cost[5][3];
```

В этом случае можно рассмотреть возможность создания нового типа и только затем объявить массивы:

```
typedef float theusual[5][3];
theusual matrix, prices, cost;
```

В этом случае вы просто создаете переменную нового типа. Компилятор будет трактовать переменную theusual как массив пять на три, элементами которого являются значения с плавающей точкой. Инструкция typedef может применяться не только к массивам.

Синтаксис этой инструкции следующий:

```
typedef объявление_типа идентификатор;
```

Самостоятельная практика

- Модифицируйте программу с7аде.cpp так, чтобы старший гимнаст называл себя. Вам потребуется сначала определить старшего. Затем этот гимнаст должен сообщить «Here!» Учтите, что вам придется определять, является ли старшим элемент Guy[0], Guy[l] и т. д., а также найти значение индекса максимального элемента. Отметьте, что решение представляет собой алгоритм поиска максимального элемента массива.



СОВЕТ

Используйте для нахождения положения старшего гимнаста целую переменную — например, oldest — и дайте ей нулевое начальное значение (это будет индекс первого гимнаста). Затем последовательно сравнивайте возраст гимнаста, на которого указывает переменная oldest, с возрастом каждого следующего гимнаста. Если возраст очередного гимнаста оказывается большим, чем у других, делайте его старшим, присвоив переменной oldest индекс этого гимнаста. Таким образом, переменная oldest всегда будет указывать на старшего из просмотренных вами гимнастов, и, когда вы просмотрите всех, она будет представлять собой индекс старшего гимнаста в группе.

- Задача поиска минимального элемента массива аналогична предыдущей. Постарайтесь найти самого молодого гимнаста.
- Напишите код функции float average(int array[], int from, int to). Возвращаемым значением функции должно быть среднее значение элементов массива, начиная с элемента с индексом from и кончая элементом с индексом to.
- Напишите код функции int less(int array[], int from, int to, int value). Возвращаемым значением функции должно быть число элементов, значения которых меньше аргумента value, начиная с элемента с индексом from и кончая элементом с индексом to. Например, если массив содержит значения 21, 32, 15, 80, 75 и 43, а вызов функции выглядит так: x=less(array, 1, 4, 50), в результате должно получиться 2.

Пример обработки числового массива

В этом примере мы будем считывать числа и сохранять их в массиве. Затем мы реализуем несколько операций, в том числе найдем максимальный элемент массива и выполним сортировку его элементов. Значения элементов массива будут выводиться в нескольких информационных рамках. Массив будет состоять из 10 целых чисел,

Заполнение и отображение массива

Простейшая версия программы будет считывать числа, сохранять их в массиве и отображать на экране в информационных рамках. Программа подразумевает выполнение следующих шагов:

- Объявление объектов и переменных.
- Ввод значений массива.
- Вывод значений массива.

Как видно из этой последовательности шагов, мы создадим функцию для ввода значений в массив и функцию для вывода этих значений на экран. Это может быть следующая программа:

```
void mainprog()
{
   int values[10];              // Массив чисел
   getarray(values, 0, 9);
   showarray(values, 0, 9);
}
```

Вся программа состоит из двух вызовов функций — вызова функции getarray() (для ввода с клавиатуры значений элементов массива) и вызова функции showarray() (для вывода этих значений на экран). Для обеих функций нужно теперь составить программы.

Реализовать такие функции можно разными способами — основная задача определить, какие типы аргументов они должны принимать.

Функция getarray()

Функция getarray() должна определять, сколько элементов вводится с клавиатуры и где их следует размещать. Можно было бы, конечно, попробовать создать функцию без аргументов и вызывать ее следующим образом:

```
getarray();
```

Эта функция будет исправно работать. В этом случае массив number можно сделать глобальным, объявив его вне основной функции mainprog(), тогда с ним смогут работать и другие функции. Можно также установить, что всегда должно считывать-

ся 10 элементов в последовательности от 0 до 9. Но такая функция вряд ли пригодится в других ситуациях. Возможно, например, вам потребуется считывать и отображать элементы других массивов, с другими именами и размерами. Итак:

- Эту функцию нельзя использовать для считывания значений в два массива разного размера.
- При использовании этой функции в другой программе необходимо будет называть массив тем же именем number. Можете сразу отказаться от попыток продать программу с такого рода недостатком!

С другой стороны, аргументы сделают вашу функцию более универсальной. Аналогичные доводы можно отнести и к функции showarray(). Основная функция программы, как вы могли убедиться, имеет простой и законченный вид. Все, что нам осталось, — это составить код для этих двух функций.

Ввод информации в массив

Функция getarray() запрашивает ввод чисел с клавиатуры и сохраняет их в массиве. Поскольку необходимо считать несколько элементов, будем пользоваться циклом. Вот возможное решение проблемы:

```
void getarray(int number[], int from, int to)
{
   for(Int i=from; i<=to; i++);
   {
      number[i]=ask("Input a number");
   }
}
```

Получилась снова очень простая функция.



ПРИМЕЧАНИЕ В C++ массивы всегда передаются функциям по ссылке, вне зависимости от того, ставите вы перед именем параметра знак & или нет. Следовательно, считанные с клавиатуры значения можно использовать в основной программе.

Вывод информации из массива

Функция showarray() выводит на экран в отдельную информационную рамку каждое значение числового массива.

Возможная реализация функции следующая:

```
void showarray(int number[], intfrom, intto)
{
   Boxlist[10];
```

```
    for(int i=from; i<=to; i++)
    {
      list[i%10].place(400, 50+(i%10)*40);
      list[i%10].label(i);
      list[i%10].say(number[i]);
    }
}
```

Поскольку диапазон индексов массива мы не контролируем, функция выводит на экран 10 рамок и размещает в них значения последних десяти элементов массива. Например, если пользователь потребует вывести элементы от 31 до 40, то элемент 31 появится в рамке 1, элемент 32 — в рамке 2 и т. д. Элемент 40 появится в рамке О, поскольку индекс массива рамок представляет собой остаток от деления на 10. На рис. 19.5 показано, как выглядят значения с 15 по 24 элемент массива.

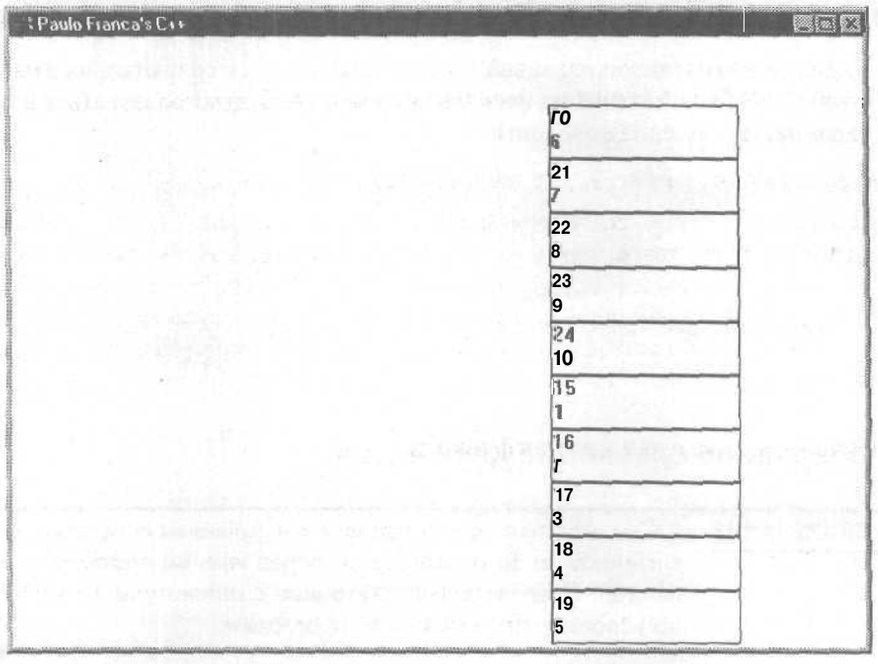


Рис. 19.5. *Вывод на экран 10 элементов массива*

В функции не используется возможность автоматического задания положения информационной рамки, то есть координаты каждой рамки задаются индивидуально. Таким образом, каждая следующая рамка оказывается под предыдущей. При каждом вызове функции создается свой набор рамок, поэтому после нескольких вызовов рамки просто перестанут помещаться на экране.

Попробуйте запустить программу.



ПРИМЕЧАНИЕ Функции, рассмотренные в этом разделе, можно найти на прилагаемой дискете в заголовочном файле c7intfun.h.

Поиск максимального элемента массива

Теперь попробуем найти максимальный элемент массива. Для ввода информации в массив и вывода информации из массива мы воспользуемся теми же функциями, что и раньше. Чтобы не выполнять поиск в основной программе, можно написать для этого специальную функцию, которая будет вызываться следующим образом:

```
findlargest(values, from, to);
```

Включение в функцию аргументов, показывающих индексы начала и конца области поиска, оказывается очень полезным. Функция должна находить *максимальное значение* или *место где оно было найдено*! Понимаете разницу между этими понятиями?

Пусть имеется следующий числовой массив:

23, 12, 45, 11, 89, 21, 32, 55, 81, 32

Просмотрев значения, вы без труда найдете максимальное, равное 89. Но вы также можете сказать, что максимальное значение находится в позиции 4 (не забывайте, что в C++ счет ведется с нуля). Если значение индекса максимального элемента известно, то его значение определить уже просто. Но с другой стороны, если вам известно максимальное значение, то определить соответствующий индекс совсем не просто. Именно по этой причине возвращаемым значением функции должен быть индекс максимального элемента, а не его значение.

Поочередное сравнение двух чисел

Как найти максимальный элемент массива?

Как вам уже хорошо известно, компьютер может одновременно сравнивать только два числа. А вот найти максимальное значение в группе чисел он самостоятельно не может. Вам требуется сообщить компьютеру, чтобы он сравнивал каждое число в группе с определенным значением. Простейшее решение проблемы состоит в том, чтобы компьютер запоминал максимальное значение, получающееся при сравнивании двух элементов массива, затем сравнивал его со следующим и т. д.

Вначале можно считать максимальным значение первого элемента массива (поскольку остальные еще не рассматривались). Затем поочередно сравнивайте его со всеми элементами и проверяйте, остается ли оно максимальным. Если это условие нарушается, замените меньшее старое значение большим новым. При этом не забывайте, что мы ищем не столько значение максимального элемента, сколько его положение в массиве (индекс).

Соответствующая реализация может иметь следующий вид:

```
int findlargest(int number[], int from, int to)
{
   int index;
   int guess;
   guess=from;                    // Первый элемент сравнения
   for(index=from; index<=to; index++)
   {
   if(number[found]<number[index])
      {
         guess=index;             // Смена элементов
      }
   }
   return guess;
}
```

Понятна работа функции? В переменной guess хранится индекс текущего максимального элемента. Вначале, поскольку ни одно число еще не проверялось, мы присваиваем ей значение первого индекса (from). Таким образом, мы считаем, что в этой позиции находится максимальный на этот момент элемент number[guess].

Далее начинается цикл. С помощью целой переменной index обозначается текущий индекс массива. Значения индекса начинаются от значения переменной from, которая обозначает позицию начала поиска, и заканчиваются значением переменной to, которая обозначает последнюю позицию поиска.



СОВЕТ Можно начать цикл с индекса from+1. Это разумно, поскольку мы уже взяли первый элемент за основу и нет никакого смысла сравнивать значения number[index] и number[found], когда индексы index и found равны. Однако в вырожденном случае, когда переменные from и to равны, это может привести к неправильному результату.

Сравнивая число, которое предполагается максимальным (number[guess]), с текущим числом (number[index]), вы уточняете положение максимального элемента. Если текущее значение оказывается большим, то старое значение индекса отбрасывается и вместо него переменной guess присваивается значение текущего индекса (index). Таким образом, гарантируется, что в переменной guess окажется индекс максимального элемента на данном этапе поиска. Следовательно, после завершения цикла значение переменной guess укажет на позицию максимального элемента массива.

Вызов функции findlargest() можно включить в основную функцию программы поиска максимального элемента массива. Разумеется, когда это число будет найдено, его следует вывести на экран. В приведенную ниже программу включены две до-

полнительные информационные рамки для вывода максимального значения и его положения в массиве.

```
void mainprog()
{
   Box largest(50, 350, "Lagest:");
   Box index(50, 400, "Pozition:");
   int values[10];
   int k;
   getarray(values, 0, 9);
   showarray(values, 0, 9);
   k=findlagest(values, 0, 9);
   lagest.say(values[k]);
   index.say(k);
}
```

Сортировка массивов

Теперь у нас есть функции для ввода значений в массив, вывода их из массива и поиска максимального элемента массива. С помощью этих функций можно также сортировать массив. Другими словами, можно расположить его элементы, например, в порядке возрастания. Сортировка представляет собой весьма распространенную и чрезвычайно полезную процедуру, заслуживающую более подробного изучения. Здесь мы ограничимся простейшим алгоритмом сортировки.

Чтобы не перегружать дополнительными инструкциями основную функцию программы, мы, как и раньше, можем создать функцию sort() и вызывать ее из основной функции программы. Это может выглядеть так:

```
sort(values, from, to);
```

Функция sort() почти не усложняет основную функцию и вдобавок может оказаться полезной в других программах!

Алгоритм функции sort() прост. Поскольку вы можете легко найти наибольший элемент, почему бы не перенести его в конец массива? Если вам известно, что наибольший элемент находится в позиции Larger, а последний в позиции last, все, что нужно сделать, — это поменять местами элементы number[larger] и number[last]. Теперь можно быть уверенным, что последний элемент размещен правильно. Что дальше? Элементы с индексами от 0 до 8 остались не рассортированными. Теперь вызываем функцию findlargest(), чтобы найти максимальный элемент в диапазоне индексов от 0 до 8 и т. д.

По мере нахождения максимальных элементов и размещения их в конце массива размер несортированного массива будет сокращаться. В конце концов мы придем к массиву из одного элемента и, таким образом, завершим сортировку!

Ниже представлена соответствующая реализация;

```
void sort(int array[], int framwhere, int towhere)
{
   int lagest, temp;
   // При каждой итерации значение переменной last уменьшается
   for (int last=towhere; last>fromwhere; last--)
   {
      // Поиск максимального алемента от начала
      // до элемента с индексом last
      largest=findlargest(array, fromwhere, last);
      // Размещение по индексу last
      // очередного максимального элемента
      temp=array[largest];
      array[largest]=array[last];
      array[last]=temp;
   }
}
```

Основная функция программы сортировки массива приобретает теперь следующий вид:

```
void mainprog()
{
   int values[10];
   getarray(values, 0, 9);
   showarray(values, 0, 9);
   sort(values, 0, 9);
   showarray(values, 0, 9);
}
```

Обмен значениями

Обмен значениями двух переменных может завести в тупик начинающего программиста. Чтобы поменять значения переменных a и b, новичок обычно пытается выполнить следующие операции:

```
a=b;
b=a;
```

Конечно, они ведут к неправильному результату. Как только переменной a будет присвоено значение переменной b, старое значение a будет потеряно. Следовательно, когда переменной b присваивается значение переменной a, в переменной b окажется новое значение a, которое равно старому значению b. Чтобы решить проблему, необходимо ввести промежуточную переменную, например temp:

```
temp=a;
a=b;
b=temp;
```

Эта проблема аналогична проблеме обмена содержимого двух стаканов — одного с вином, а другого с молоком. Для этого обязательно нужен третий стакан. Сначала переливаем в него вино, затем переливаем молоко в стакан из-под вина. И лишь после этого вино можно перелить в стакан из-под молока.

Рекурсивная сортировка

Возможен следующий вариант функции sort(): сначала проверяется не состоит ли массив всего из одного элемента. Если да, то сортировка закончена. Если нет, ищется максимальный элемент и меняется местами с элементом, находящимся в конце массива. Затем сортируются оставшиеся элементы.

Что означает *сортируются оставшиеся элементы?* Это означает, что функция sort() вызывается снова. Таким образом, данное решение рекурсивно, поскольку функция sort() вызывает саму себя.

Один из возможных вариантов рекурсивной функции сортировки может быть реализован следующим образом:

```
void recsort(int array[], int fromwhere, int towhere)
{
   // Рекурсивная версия сортировки
   int largest, temp;
   int last;
   last=towhere;
   // Проверка окончания сортировки
   if(fromwhere<=towhere)
   {
      // Поиск максимального элемента от начала
      // до элемента с индексом last
      largest=findlagest(array, fromwhere, last);
      // Размещение по индексу last
      // очередного максимального элемента
      temp=array[largest];
      array[largest]= array[last];
      array[last]=temp;
      // Сортировка оставшейся части массива
      recsort(array, fromwhere, last-1);
   }
}
```

Что нового мы узнали?

В этом уроке мы научились

- Обрабатывать однотипные объекты с помощью массива.
- Работать с массивом объектов или чисел.
- Использовать массив массивов.
- Сортировать массив.



УРОК 20 Символьные массивы

- Использование символьных массивов
- Обработка строк с помощью функций заголовочного файла **string.h**
- Преобразование символов в числа, и наоборот
- Использование структур
- Поиск в массиве

В C++ текст можно обрабатывать в символьных массивах, где каждый элемент представляет собой символ. Для большинства обычных операций с символьными массивами имеются стандартные функции. Вам не придется самому программировать сравнение строк или копирование одной строки в другую,

Иногда информацию требуется представить в виде отдельных составляющих, например номера счета, имени покупателя и его текущего баланса. Эта задача может быть решена не только с помощью объектов, но и с помощью структур.

Если данные хранятся в массиве, возникает задача нахождения заданного элемента массива. Как реализовать такой поиск, вы узнаете в конце этого урока.

Текст в программах на C++

Еще одним важным приложением компьютеров является обработка текста. Хотя компьютер хранит в своей памяти только числа, посредством числовых кодов он может также хранить и обрабатывать символы. При этом в большинстве случаев код, которым представлены символы, значения не имеет. Вы сможете научиться работать с символами, даже не зная их кодов.

Тип char

Тип char могут иметь переменные, предназначенные для хранения символов. Например, в следующей инструкции для хранения символа объявляется переменная initial:

```
char initial;
```

В каждой переменной типа char может храниться только один символ. По этой причине обрабатывать символы удобнее с помощью массивов. Символьную переменную можно использовать в выражении, а также присваивать ей символьное значение, заключая соответствующий символ в одинарные кавычки. Например, в следующей инструкции переменной initial присваивается значение, соответствующее символу A (в верхнем регистре):

```
initial = A;
```

Начальные значения можно задавать при объявлении переменных. Например:

```
char choice = y;
```

Необходимо помнить, что прописные и строчные буквы имеют разный код. Программа может отличить символ a от символа A.

Сравнение кода символов

Разумеется, символьные переменные, как и всякие другие, можно сравнивать между собой:

```
char choice;
...
if (choice==N) break;
```

В рассмотренном примере символьная переменная choice сравнивается с символом N (в верхнем регистре). Если в переменной choice окажется код символа N, выполняется инструкция break.

Пробел

Для обозначения пробела его также необходимо поместить внутрь кавычек. Наряду с остальными символами пробел тоже кодируется:

```
char blank= ;
```

В этой инструкции объявляется переменная blank типа char, и этой переменной присваивается пробел в качестве начального значения.

Символы и их коды

В C++ символьной переменной может быть присвоено числовое значение (int, Long, float и т. д.), поскольку представляющий символьную переменную код является целым числом в интервале от 0 до 255. На самом деле одинарные кавычки, в которые заключается символ, и обозначают числовой код этого символа. Другими словами, 'N' — это то же, что 78, 'A' — 65, 'a' — 97 и т. д. Не стоит, однако, слишком задумываться о числовых кодах — это стандартные коды для передачи символов по электронным коммуникациям. Чуть больше информации по данной теме вы получите из упражнений этого урока. Если числовое значение окажется вне интервала от 0 до 255, компьютер отбросит лишние разряды и результат будет не тем, что вы ожидали.

Можно также присвоить символ целой переменной. Однако присваивать символьной переменной более одного символа (символьный массив) недопустимо.

Ячейка памяти, отводимая компьютером для хранения символа, может хранить числовой код в диапазоне от 0 до 255. Таким образом, всегда может быть представлено 256 символов. Этого достаточно, чтобы обозначить все буквы латинского ал-

фавита в верхнем и нижнем регистре, а также все цифры, специальные знаки и управляющие символы. Все, что нам нужно сделать, — это связать код символа с самим символом. В большинстве компьютеров используется так называемый код ASCII (American Standard Code for Information Interchange, Американский стандартный код для обмена информацией).

Самостоятельная практика

О Допустим, что в программе сделаны следующие объявления:

```
int i, j, k;
float x, y;
char a, b;
```

- Какие из следующих инструкций правильны с точки зрения синтаксиса:

```
i=j;
a=j;
x=a;
for(a=a; a<=z; a++)
b=b+3;
```

Использование символьных массивов

Поскольку каждая переменная может содержать в себе только один символ, то единственный способ обращаться со словами и целыми фразами состоит в использовании символьных массивов. Б некоторых языках программирования имеется специальный тип данных (так называемый строковый тип данных) для обработки последовательностей символов — *строк* (strings). Однако не забывайте, что в C++ имеются классы. Для обработки строк вы можете создать свой класс или приобрести его у сторонних производителей!

Символьные массивы объявляются так же, как и все остальные. Например:

```
char first_name[20];
```

С каждым элементом такого массива можно работать. Например:

```
first_name[0]=a;
first_name[1]=n;
first_name[2]=first_name[0];
```

В результате выполнения приведенной выше последовательности инструкций в первых трех позициях массива first_name будет храниться идентификатор ana.

При объявлении символьного массива его также можно инициализировать:

```
char last_name[20]= {f, r, a, n, c, a};
```

Контроль длины слова

Поскольку слова и имена имеют разные длины и поскольку каждый адрес в памяти либо инициализирован некоторым значением, либо содержит мусор, как мы узнаем, что, например, данная фамилия состоит из шести символов? Очевидно, если соответствующая переменная была создана, чтобы хранить только одно это имя, мы могли бы запомнить его размер — хотя обычно поступают по-другому. Знать, где заканчивается строка, весьма полезно.

Хотя в C++ нет строкового типа данных, но есть некоторые функции, призванные помочь программисту при работе со строками. По соглашению строки в C++ завершаются специальным символом, который находится в памяти за последним символом строки. Этот специальный символ называется символом null. Это не ключевое слово, хотя оно определено почти в каждом заголовочном файле C++. Фактически, символ завершения строки, хотя и называется символом, определяется как число О (но не как символ 0).

Чтобы правильно инициализировать массив last_name в последнем примере, можно выбрать один из следующих вариантов:

```
char last_name[20]={f, г, а, п, с, а, null};
char last_name[20]={f, г, а, п, с, а, 0};
```



ПРИМЕЧАНИЕ Ноль не заключается в одинарные кавычки, поскольку мы имеем в виду *нулевое* значение, а не символ *0*.

Приведенные примеры инициализации символьного массива не слишком удобны, но в C++ имеется другой путь:

```
char last_name[20]="franca";
```

Эта инструкция работает так же, как и предыдущие. Когда вы заключаете последовательность символов в двойные кавычки, вы даете знать компилятору, что описываете строку. Компилятор автоматически добавляет null в конец строки при условии, что в массиве достаточно места для его размещения. Последовательность символов, в конце которой стоит символ null, называется *оканчивающейся нулем строкой* (null-terminated string).

Как вы наверное помните, сообщения наших гимнастов мы заключали в двойные кавычки. Почему? Потому что последовательность символов, заключенная в двойные кавычки, — это оканчивающаяся нулем строка. Как правило, можно заменить любое сообщение, заключенное в двойные кавычки, любой оканчивающейся нулем строкой.

Например, можно написать следующую инструкцию:

```
Sal.say(last_name);
```

Аналогично, оканчивающиеся нулем строки можно использовать в качестве меток информационных рамок или в качестве запросов на ввод данных с клавиатуры.

Строковые массивы

В C++ можно обрабатывать массив, каждый элемент которого представляет собой символьный массив, то есть массив массивов. В следующей инструкции объявляется массив из пяти элементов, каждый из которых представляет собой массив из 20 символов:

```
char name [5][20];
```

Массив такого типа может пригодиться, например, для хранения списка имен. Обратите внимание, что элемент name[0], равно как и элементы name[1], name[2], name[3] и name[4], представляет собой символьный массив. В каждом таком массиве могут храниться оканчивающиеся нулем строки. Например, в следующем цикле считывается по одному имени в каждый из пяти массивов из 20 символов:

```
for (int i=0; i<=4; i++)
   askwords(name[i], 20, "Enter a name");
```

Предположим, мы хотим хранить имена пяти гимнастов. Для этого нужен массив из пяти элементов и массив имен. Можно хранить имя первого гимнаста в массиве name[0], имя второго — в массиве name[1] и т. д.

Более читабельная версия той же программы выглядит следующим образом:

```
typedef char onename[20];          // Создание типа onename
onename namelist[5];               // Массив namelist типа onename
                                   // содержит 5 имен по 20 символов
for (int i=0; i<=4; i++)
   askwords(namelist[i], 20, "Enter a name");
```

Можно также инициализировать строку без указания длины. Например,

```
char first_name []="Mitiko";
```

Даже если в скобках [] ничего нет, их нужно все равно указывать,

В этом случае объявляется символьный массив first_name размером 7 символов (шесть плюс null). Это удобно, потому что избавляет от необходимости подсчитывать размер строки. Тем не менее помните, что в этом случае в массиве нельзя сохранить имя, длиннее шести символов.

Ввод строк

Для ввода строки с клавиатуры в символьный массив можно использовать функцию askwords{), включенную в заголовочный файл franca.h. Эта функция отображает диалоговое окно с сообщением, указанным в качестве третьего аргумента:

```
askwords(char inputstring[], int maxsize, char message[])
```

В этом диалоговом окне пользователь может ввести строку, которая заменит первый аргумент и размер которой указывается в качестве второго аргумента.

Например, программа ниже считывает список имен и отображает соответствующее имя под каждым гимнастом.

```
#include "franca.h"
void mainprog()          // c7names.cpp
{
   athlete player[5];
   char name [5][20];
   for (int i=0; i<=4; i++)
      askwords(name[i], 20, "Enter a name");
   for (i=0; i<=4; i++)
   {
      player[i].ready(0.);
      player[i].say(name[i]);
   }
}
```

Строковые функции

К сожалению, в отличие от числовых переменных, строки нельзя сравнивать и копировать друг в друга с помощью оператора присваивания. Например, пусть программа содержит следующие объявления:

```
char agent[]="Bond", client[]="Flint";
```

Тогда представленные ниже инструкции работать не будут:

```
if(agent=="Bond")...
   client=agent;
```

Тем не менее для решения этой задачи нет необходимости поочередно сравнивать или копировать все элементы символьного массива. Необходимые функции находятся в заголовочном файле string.h. Это стандартный заголовочный файл, доступный почти в каждом компиляторе C++. Доступ к строковым функциям можно получить с помощью директивы #include:

```
#include <string.h>
```

Мы не станем подробно останавливаться на всех функциях заголовочного файла string.h, а рассмотрим только наиболее популярные. Дополнительную информацию о строковых функциях можно найти в системе контекстной помощи вашего компилятора. Мы ограничимся следующими функциями:

- Функция strcmp() сравнивает две строки (в соответствии с алфавитом).
- Функция stricm() сравнивает две строки (в соответствии с алфавитом) без учета регистра символов.

- Функция strcpy() копирует строку в другую строку.
- Функция strcat() добавляет строку в конец другой строки.
- Функция strlen() вычисляет длину строки.

Сравнение строк

Следующая функция сравнивает символьные массивы s1 и s2:

```
int strcmp (char s[1], char s2[]);
```

Если в соответствии с алфавитом строка s1 оказывается перед строкой s2, функция возвращает отрицательное целое. Если обе строки в соответствии с алфавитом идентичны, функция возвращает 0. Если в соответствии с алфавитом строка s1 оказывается после строки s2, функция возвращает положительное целое.

Например:

```
char city[20];
Cin>>city;
if (strcmp(city, "Cupertino")==0)
   Cout<<"You live in good town!";
```

В приведенном выше примере значение строки city вводится с клавиатуры и затем сравнивается со строкой Cupertino. Если строки идентичны, на экран выводится сообщение You Live in good town!. Однако, если вы введете, например, cupertino, соответствие будет не полным, поскольку регистр строк разный. Для решения такого рода проблем используется описанная ниже функция stricmp().

Сравнение строк без учета регистра символов

Следующая функция сравнивает символьные массивы s1 и s2:

```
int stricmp(char s1[], char s2[]);
```

Эта функция работает аналогично функции strcmp(), но регистр символов игнорируется.

Копирование строк

Следующая функция копирует исходную строку (параметр source) в целевую строку (параметр dest):

```
strcpy (char dest[], char source[]);
```

При копировании исходная целевая строка теряется. Убедиться в том, что исходная строка уместится в целевой строке, является обязанностью программиста.

Пример:

```
char name1[]="Brandon", name2[]="Daisy";
strcpy(name1, name2);
...
```

В этом примере массив name1 после вызова функции будет содержать строку Daisy. Обратная ситуация (копирование name1 в name2) ведет к ошибке, поскольку размер массива name2 недостаточен, чтобы вместить строку из массива name1.

Конкатенация строк

Следующая функция добавляет исходную строку (параметр source) к концу целевой строки (параметр dest):

```
strcat (char dest[], char source[]);
```

Пример:

```
char message[20]="The exit "
strcpy(message, "is near!");
...
```

После выполнения этого фрагмента кода символьный массив message будет содержать строку The exit is near!.

Определение длины строки

Следующая функция возвращает длину строки:

```
int strlen(char string[]);
```

Длина рассчитывается без учета символа завершения строки (null).

Пример:

```
char sentence[80]="He that shall persevere to the end ...";
Cout << strlen(sentence);
...
```

В этом примере на экране появится число 38, равное числу символов в массиве sentence. Таким образом, длина строки и размер массива — это не одно и то же! Длина строки — это число символов от начала строки до символа завершения строки (null). Эта длина вполне вероятно будет меняться по ходу выполнения программы. С другой стороны, размер массива обычно задается при объявлении массива. Длина строки никогда не может и не должна быть больше размера массива.

Оператор sizeof()

Хотя размер массива не меняется при выполнении программы, в C++ имеется оператор для определения объявленного в программе размера массива, объекта или структуры. Это оператор sizeof().

Оператор sizeof() возвращает целое, показывающее размер любой объявленной в программе переменной, объекта, массива или структуры. Очевидно, размер переменной можно определить при ее объявлении. Для чего тогда нужен этот оператор?

Есть несколько причин, по которым он может оказаться полезным. В частности, кажущиеся очевидными вычисления довольно часто становятся утомительными. Например, каков размер массива в следующем объявлении?

```
char listen[]="He that shall persevere to the end, he shall be saved.";
```

Сосчитать количество символов не представляет проблем. Просто сложите все буквы, пробелы, точку и, наконец, не забудьте добавить еще единицу, чтобы учесть символ завершения строки. Однако это не только утомительно, но и может вести к ошибкам.

Другая, более важная причина состоит в том, что вам часто придется изменять программу, и массив, который был сначала объявлен с 40 элементами, в окончательной версии может содержать 50. Если у вас имеется цикл от 0 до 39 для обработки массива, вам придется просмотреть всю программу и найти константы, требующие изменения. Например:

```
char array[40];
...
for (int i=0; i<=39; i++)
```

Если заменить этот фрагмент следующим, можно легко избежать проблем:

```
char array[40];
...
for (int i=0; i<sizeof(array); i++)
```

Преобразование символов в числа, и наоборот

Каждая цифра — 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9 — может быть представлена в виде символа, поэтому любое число, хранящееся в компьютере, может быть заменено набором цифр (символов), и наоборот. В C++ для выполнения таких преобразований имеются следующие функции;

- Функция atoi() преобразует алфавитно-цифровой (символьный) массив в целое.
- Функция atof() преобразует алфавитно-цифровой (символьный) массив в число с плавающей точкой.
- Функция itoa() преобразует целое в алфавитно-цифровой (символьный) массив.

Эти функции находятся в заголовочном файле stdlib.h, который включается в программу следующей директивой:

```
#include<stdlib.h>
```

Преобразование символьного массива в целое

Следующая функция возвращает целое, которое представлено в символьном массиве (строке) number:

```
int atoi(char number[])
```

Пример:

```
char number[20];
int k;
askwords(number, 20, "Enter the value");
k=atoi(number);
Cout<<k;
```

Если массив невозможно преобразовать или он содержит недопустимые символы, функция возвращает ноль.

Преобразование символьного массива в число с плавающей точкой

Следующая функция возвращает число с плавающей точкой, которое представлено в символьном массиве (строке) number:

```
float atof(char number[])
```

Если массив невозможно преобразовать или он содержит недопустимые символы, функция возвращает ноль.

Преобразование целого в символьный массив

Следующая функция преобразует целое значение value в символьный массив и сохраняет результат в строке number:

```
itoa(int value, char number[], int radix)
```

Целым значением radix обозначается основание системы счисления. Поскольку обычно мы работаем в десятичной системе счисления, это значение равно 10.

Пример:

```
int k=25;
char number[20];
itoa(k, number, 10);
```

Самостоятельная практика

- Составьте программу, которая должна считывать ваше имя, отчество и фамилию. Затем она должна вывести на экран фамилию, а через запятую имя и отчество.
- Составьте функцию, которая должна сравнивать, начиная с позиции i и кончая позицией j, один символьный массив с другим, начиная с позиции k. Возвращаемым значением функции должна быть 1, если строки идентичны, и 0 в противном случае.

- Составьте программу, которая должна сначала считать целое число N, а затем считать N имен, поочередно сохраняя их в массиве символьных массивов.
- Усовершенствуйте предыдущую программу так, чтобы она могла считывать целое число m и затем отображать на экране элемент массива с этим номером.

Структуры

Структурой (structure) называется группа данных различных типов и/или назначения, которые представляют собой единый информационный элемент. Например, к покупателю может относиться следующая информация:

Имя

Адрес

Индекс

Номер телефона

Номер счета

Каждый элемент данных, называемый *полем* (field), имеет различное назначение. Одно поле содержит имя, другое название улицы, третье номер телефона и т. д. Однако все эти поля связаны между собой, поскольку относятся к одному и тому же покупателю, и обычно называются *записью* (record). Поле адреса содержит адрес покупателя, фамилия которого указана в поле имени, поле с телефонным номером — телефонный номер и т. д.

Поскольку в объектах также можно хранить группы разнотипных данных, можно сказать, что они тоже содержат структуры. Например, гимнасты обладают координатами, равно как головой, телом, руками и ногами. Фактически между структурами и классами имеется очень большое сходство. В структуру можно включить функции-члены и затем объявлять объекты этой структуры. Единственное отличие структуры от класса в том, что в структуре нет закрытых и защищенных членов. Все члены структуры открытые.



ПРИМЕЧАНИЕ Хотя классы можно заменить структурами, я рекомендую использовать структуры только в тех случаях, когда в вашей программе отсутствует необходимость в функциях-членах.

Объявление структуры аналогично объявлению класса, но вместо ключевого слова class ставится ключевое слово struct:

```
struct идентификатор
{
   Объявления данных или объектов
};              // Не забывайте про точку с запятой!
```

Например:

```
struct student
{
   char lastname[20];
   char firstname[20];
   long enrollment_number;
};
```

Как и в случае с классами, объявление структуры не подразумевает создание объекта или переменной. Объявление — это просто описание будущего объекта, в данном случае студента. Чтобы использовать объект определенного класса или переменную определенного типа их сначала необходимо объявить в качестве объекта этого класса или переменной этого типа. Аналогичным образом, для использования структурной переменной необходимо объявить переменную этой структуры. Например, следующая инструкция создает переменную customer структуры student:

```
student customer;
```

Доступ к полям структуры задается именем соответствующей структурной переменной. Например:

```
customer.enrollment_number=924054;
```

Как вы помните, доступ к данным объекта осуществляется тем же самым способом.

Структуры и массивы

Поле структуры может быть массивом. Так, в структуре student полями firstname и lastname являются символьные массивы. Поле структуры может быть другой структурой. Рассмотрим следующую структуру:

```
struct adress
{
   char street[40];
   char city [20];
   char zipcode[10];
};
```

Объявив эту структуру адресов, мы можем затем объявить структуру сотрудников и включить туда поле, представляющее собой структуру адресов. Другими словами, мы получаем одну структуру внутри другой. Например:

```
struct employee
{
   char lastname[20];
   char firstname[20];
```

продолжение

```
    long salary;
    address home;
}
```

Если вас интересует доступ к полю адресов, то ниже показано, как это можно сделать:

```
employee person;
...
strcpy(person.home.street, "221 Baker St.");
strcpy(person.home.city, "London");
...
```

Наследование в структурах

Другим способом создания рассмотренной выше структуры employee является ее наследование из структуры address. Иначе говоря, сотрудника можно выразить через адрес, имя, отчество и зарплату.

```
struct employee:address
{
    char lastname[20];
    char firstnane[20];
    long salary;
};
```

Массивы структур

Возможно создание массивов, элементами которых являются структуры. Например:

```
employee staff[100];
student group[30];
```

Первая инструкция создает массив из 100 элементов, каждый из которых обозначает сотрудника вместе со всеми полями, присущими элементу структуры employee. Вторая инструкция создает массив из 30 студентов.

Для доступа к полю city данного сотрудника необходимо указать соответствующий индекс массива, чтобы компьютер мог понять, о ком идет речь. Например:

```
staff[10].home.city
```

Копирование структуры

Используя оператор присваивания, можно копировать одну переменную в другую, имеющую ту же структуру. Например, пусть заданы следующие объявления:

```
employe worker1, worker2, staff[100];
```

Тогда, поскольку компилятор может копировать переменные с одинаковой структурой, допустимы следующие инструкции:

```
worker1=worker2;
staff[5]=worker1;
worker1.home=staff[1].home;
```

Поиск в символьном массиве

Одним из интересных приложений поиска заданного элемента в массиве является поиск в символьном массиве. Например, вам может понадобиться узнать, какой элемент массива имеет нулевое значение.

Эту ситуацию можно проиллюстрировать на примере оканчивающейся нулем строки. Помните, что символом завершения строки является null, значение которого равно нулю? Как вы думаете, как функция strlen() вычисляет длину строки?

Все, что вам нужно, — это просматривать элементы массива, пока не встретится символ null. Это легче, чем найти максимальный элемент в массиве, хотя бы потому, что вы знаете значение элемента, который хотите найти.

Итак, вы сравниваете искомое значение с первым элементом, со вторым и т. д.

Пусть у вас есть следующая строка:

```
char name[50];
```

Пусть теперь вам требуется определить длину строки без использования функции strlen(). Бы можете решить эту задачу следующим образом:

```
for(int position=0; position<50; position++)
{
   if (name[position]==0) break;
}
```

После выполнения этого фрагмента программы переменная position должна содержать число символов перед символом завершения строки. Но если этот символ не найден, ответом будет число 50.



ПРИМЕЧАНИЕ Теперь вы знаете, как найти симол null, а значит, сможете найти любой другой символ, например t.

Поиск в многомерных массивах

Еще более интересная ситуация возникает при поиске в структуре или многомерном массиве. Например, рассмотрим следующую структуру:

```
struct student
{
   int  idnumber;
  char lastname[30];
}
student myclass[50];
```

В этом случае у нас есть массив myclass с 50 элементами. Каждый элемент массива — это структура, состоящая из идентификатора и фамилии.

Номер элемента	Идентификатор	Фамилия
00	5002	Rogers
01	6754	Smith
02	6003	Adams
03	6532	Rodriguez
19	6021	Arentz

Предположим, что все элементы массива содержат допустимые данные. Как теперь найти фамилию студента с идентификатором 6021?

Эта задача в общем-то та же, что и при поиске нужного значения в массиве. Нужно просто искать соответствующее значение в поле idnumber. После того как значение найдено, нужно сохранить номер текущего элемента массива:

```
for (int position=0; position<50; position++)
{
   if(myclass[position].idnumber==6021) break;   // Готово!
}
```

Если найдено соответствие, например, в позиции 19 — фамилия, которая вам нужна, находится в поле lastname позиции 19 в массиве. То есть вы можете получить фамилию следующим образом:

```
myclass[position].lastname
```

Крах поиска

Что если в списке нет студента с указанным идентификатором? В этом случае цикл пройдет по всем элементам массива, а инструкция break выполнена не будет. Тем не менее какое-то значение в переменной position останется. Но неправильное. Чтобы избежать этого, контролируйте значение переменной position, которое должно лежать в интервале от 0 до 49. После неудачного завершения цикла в переменной position окажется значение 50.



СОВЕТ Полезно создать для поиска специальные функции. В этом случае можно возвратить индекс, если соответствие найдено, и отрицательное число в противном случае.

Самостоятельная практика

- Напишите код функции с параметрами inputstring и thischaracter. Параметр inputstring — это строка, a thischaracter — символ. Возвращаемым значением функции должна быть позиция, найденная в результате поиска символа thischaracter в строке inputstring. Если символ не найден, функция должна возвращать нулевое значение.
- Напишите программу, которая должна считывать с клавиатуры массив из 10 целых значений и определять позицию заданного значения. Это значение также должно вводиться с клавиатуры. Для поиска в массиве используйте функцию.

Что нового мы узнали?

В этом уроке мы научились

- ☑ Использовать символьные массивы для хранения текстовых данных.
- 0 В операциях с символьными массивами использовать функции обработки строк.
- 0 Преобразовывать символы в числа и наоборот.
- 0 Использовать структуры.
- 0 Выполнять поиск в массивах заданных значений.



УРОК 21 Разработка приложений

- Совершенствование торгового терминала
- Совершенствование спутниковой системы

В русле дальнейшего развития навыков создания приложений мы модифицируем два созданных ранее приложения — проект торгового терминала (созданного впервые на уроке 9) и проект спутниковой системы (созданного на уроке 18). Изменения будут следующими:

- В проекте торгового терминала появятся массивы и обработка текста.
- В проекте спутниковой системы массивы помогут нам обрабатывать группу спутников.

Проект торгового терминала

Тот, кто покупал продукты в большом магазине, мог заметить, что кассир обычно не знает их цену. Он либо видит цену на наклеенном на упаковке ярлыке, либо, как это делается в современных супермаркетах, считывает штрих-код с помощью специального устройства. Как в последнем случае работает торговый терминал?

В памяти компьютера содержится список всех предназначенных для продажи товаров. В этот список входит код, стоимость и иногда краткое описание товара. По введенному кассиром коду компьютер находит в этом списке нужный товар. Не напоминает ли это вам операцию поиска в массиве структур? Конечно, это она!

В нашей новой версии торгового терминала каждая покупка будет начинаться с ввода с клавиатуры (заменяющей нам считывающее устройство) штрих-кода товара. Затем мы найдем соответствующую ему позицию и выведем на экран цену и краткое описание товара.

При этом торговый терминал конечного пользователя должен, по возможности, остаться прежним.

Реализация новых возможностей

В памяти компьютера должна храниться следующая информация о каждом продукте:

- Код
- Описание
- Цена

Поскольку товаров много, можно объявить массив структур:

```
struct  product
{
   int code;
   char description[20];
   float price;
};
productlist[20];
```

Используйте этот массив для хранения информации по всем интересующим вас пунктам. Поскольку оперативная память компьютера не может сохранять информацию при выключенном питании, вам придется заполнять массив при каждом запуске программы. Этот массив можно представить себе в виде каталога (списка) атрибутов товара, который проверяется при каждой покупке.

Для поиска в каталоге информации о товарах можно объявить класс catalog. С помощью этого класса мы будем находить информацию по коду товара. При наличии такого каталога оставшаяся часть операций будет не слишком отличаться от работы старого терминала. Но действительно ли нам нужно создавать новый класс? Ведь многое можно унаследовать из старого!

Класс catalog

Класс catalog должен выполнять все операции, необходимые для работы с каталогом товаров. Хотя эта частная реализация решения всего одной задачи, с которой нам пришлось столкнуться, позже вы поймете, насколько полезен этот фрагмент кода. Вот возможное объявление:

```
struct product
{
   int code;
   char description[20];
   float price;
};
class catalog
{
protected:
   product list[20];
   int listsize;
public:
   catalog();
   virtual product find(int part_number);
}
```

Данными-членами этого класса являются массив структур и целое значение, обозначающее число элементов массива (число товаров). Каждый элемент массива — это структура, в которой хранятся атрибуты каждого товара.

Что же касается функций-членов, то это конструктор и функция find(). Конструктор будет автоматически запрашивать ввод элементов массива, чтобы вы не могли использовать пустой массив. Функция find() будет выполнять поиск заданного товара по его коду.



ПРИМЕЧАНИЕ В производных классах функцию find(), возможно, потребуется заменять другими одноименными функциями. Поэтому она должна быть объявлена виртуальной функцией.

Программный код конструктора:

```
catalog::catalogC)
{
   listsize=0;
   for(listsize=0; yesno("Another item?")&&(listsize<20); listsize++);
   {
      list[listsize].code=ask("Enter product code:");
      strcpy( list[listsize].description, "Product decription");
      askwords(list[listsize].description, 20, "Description:");
      list[listsize].price=ask("Enter the price:");
   }
}
```

Функция findQ очень похожа на ту, с которой мы имели дело при поиске в массиве:

```
product catalog::find(int part_number)
{
   for(int item=0; item<listsize; item++);
   {
      if(part_number==list[item.code) return list[item];
   }
   product nonexistent;
   nonexistent.code=0;
   return nonexistent;
}
```

Если запрошенный код не совпадет ни с одним из представленных в списке, будет возвращена структура с нулевым кодом, и это будет свидетельствовать о том, что данный пункт в списке не найден. С другой стороны, если совпадающий код будет найден в позиции item, то возвращаемая структура будет представлять собой элемент массива с индексом item.

ПРИМЕЧАНИЕ Код класса catalog находится в заголовочном файле c7catalo.h на прилагаемой дискете.

Новый класс saleterm

Новый класс saleterm является производным от старого класса terminal. Унаследовав все необходимые атрибуты старого класса, он просто обеспечивает его дополнительными возможностями. Класс может быть объявлен следующим образом:

```
class saleterm:public terminal
{
protected;
   catalog parts;
   product sale;
   Box Part_Number, Part_Description;
   void items();
public:
   saleterm();
};
```

Создавая производный от класса terminal класс, вы наследуете все, что он содержал. Необходимо только объявить новые элементы. Новый терминал состоит из каталога, структуры для описания одного продукта (того, который в данный момент продается), пары новых информационных рамок для отображения кода продукта (Part_Number) и описания продукта (Part_Description).

Функция items() тоже наследуется. Однако, поскольку теперь вместо цены терминал запрашивает код, нам нельзя использовать ту же самую функцию. Необходимо создать новую, предназначенную для обработки кодов.

Функцию-член operate() *можно* оставить, унаследовав ее от класса terminal, поскольку она начинает действовать уже после того, как получено значение цены.

Конструкторы

Интересный случай возникает, когда с помощью конструкторов создается объект производного класса. (Внимание — это для вас ново!) Сначала вызывается конструктор базового класса, а затем производного. При этом, когда вызывается объект производного класса, соответствующий объект базового уже существует. Этот новый конструктор лишь добавляет объекту атрибуты, характерные для производного класса.

В данном примере конструктор нужен только для того, чтобы разместить две дополнительные информационные рамки и присвоить им метки.

Ниже представлен программный код конструктора:

```
saleterm::saleterm()
{
   Part_Number.place(450, 300);
   Part_Description.place(450, 340);
   Part_Number.label("Part_Number:");
Part_Description.label("Description:");
}
```

Код новой функции items():

```
void saleterm::items()
{
   int somecode;
   for(;;)
   {
      do
      {
         somecode=ask("Enter part number:");
         sale=part.find(somecode);
         if(sale.code==0) yesno("Wrong part number, please check");
       }
      while (sale.code==0);
      Part_Number.say(sale.code);
      Part_Description.say(sale.description);
      saletotal=saletotal+sale.price;
      Price.say(sale.price);
      Cur_Total.say(saletotal);
      if(!yesno("Another item?");
    }
   saletotal=saletotal*(1+tax);
   Saletotal.say(saletotal);
}
```

Полный код этого проекта, который можно найти в файлах c7term.h и c7term.cpp на прилагаемой дискете, представлен ниже:

```
#ifndef C7TERM_H          //c7term.h
ftdefineC7TERM_H
#include "franca.h"
#include "c6term.h"
```

продолжение

```
#include "c7catalo.h"
#include <string.h>
class saleterm:public terminal
{
protected:
   catalog parts;
   product sale;
   Box Part_Number, Part_Discription;
   void items();
public:
   saleterm();
};
saleterm::saleterm()
{
   Part_Number.place(450, 300);
   Part_Description.place(450, 340);
   Part_Number.label("Part_Number:");
   Part_Description.label("Description:");
}
void saleterm::items()
{
   int somecode;
   for(;;)
   {
      do
      {
         somecode=ask("Enter part number: ");
         sale=part.find(somecode);
         if(sale.code==0) yesno("Wrong part number, please check");
      }
      while (sale.code==0);
      Part_Number.say(sale.code);
      Part_Description.say(sale.description);
      saletotal=saletotal+sale.price;
      Price.say(sale.price);
      Cur_Total.say(saletotal);
      if(!yesno("Another item?");
   }
   saletotal=saletotal*(1+tax);
   Saletotal.say(saletotal) ;
}
#endif
```

```
// Конец кода заголовочного файла

#include "franca.h"          // c7term.cpp
#include "c7term.h"
ffinclude <string.h>
void mainprog()
{
   saleterm cashregister;
   cashregister.operate();
}
```



ПРИМЕЧАНИЕ Можете ли вы так сохранить информацию о продукте, чтобы не вводить ее заново при каждом запуске программы? Иначе программное обеспечение для вашего терминала вряд ли удастся продать, не так ли? Как мы узнаем на следующем уроке, эту информацию можно сохранить на дисковом файле.

Проект спутниковой системы

Массивы позволяют легко обрабатывать большие количества объектов. Так, если вы решите имитировать вращение электронов вокруг неподвижного ядра методами предложенными на уроке 18, то есть без использования массивов, то получите очень сложную программу с множеством циклов. Каждый электрон в такой программе будет обрабатываться индивидуально.

Поскольку массивы позволяют обращаться сразу ко всем объектам по одному имени (например, electron [n]), для их обработки достаточно написать один цикл. Такая программа не только экономит ресурсы, но не меняется при изменении числа элементов массива.

В качестве примера имитации атома с двумя электронами на первой орбите и восемью на второй мы используем класс satellite и массив electron, состоящий из 10 объектов типа satellite. Как и раньше, само ядро тоже можно считать спутником (объектом типа satellite).

Расширение возможностей модели с помощью массивов

Чтобы модель внутреннего космоса была интересней, можно заставить ядро тоже вращаться по орбите. В приведенном ниже примере (программа c7atom.cpp) в качестве центра ядра используется спутник ether.

Объявления и инициализация приведены ниже:

```
const float pi2=2*3.14159;
Box clock("Time:");
Clock timer, sidereal;
satell ite electron[10];
satellite nucleus, ether;
nucleus.resize(40);
ether.place(320, 200);
nucleus.center(ether);
nucleus.dist(80);
nucleus.speed(pi2/800.);
nucleus.move();
```

Инициализация электронов

Нужно инициализировать также каждый электрон.

Инициализация электронов на первой орбите:

```
for (int i=0; i<2; i++)
{
  electron[i].center(nucleus);
  electron[i].dist(80);
  electron[i].speed(pi2/300.);
  electron[i].resize(12);
  electron[i].color(5, 5);
  electron[i].angle(i*pi2/2.+pi2/4.);
  electron[i].move();
}
```

Инициализация электронов на второй орбите;

```
for (int i=0; i<10; i++)
{
  electron[i].center(nucleus);
  electron[i].dist(120);
  electron[i].speed(pi2/600.);
  electron[i].resize(12);
  electron[i].color(5, 5);
  electron[i].angle(i*pi2/8.);
  electron[i].move();
}
```

После того как все объекты будут инициализированы, начинается собственно моделирование:

```
nucleus.show();
for(; sidereal.time()<20.;)
```

```
{
   nucleus.erase();
   nucleus.move();
   nucleus.show();
   for(int i=0; i<10; i++);
   {
      electron[i].erase();
      electron[i].center(nucleus);
      electron[i].move();
      electron[i].show();
   }
   clock.say(sidereal.time()*10);
   timer.watch(.033);
   timer, reset();
}
```

Б данном случае ядро было представлено в виде отдельного спутника, но представлять его именно так не обязательно. Как уже говорилось, ядро можно считать одним из спутников, то есть нужно добавить в массив еще один электрон electron [10].

Самостоятельная практика

- В представленной ниже программе c7pool.cpp реализован класс pool, который имитирует шар, катающийся по бильярдному столу и сталкивающийся с его стенками.

  * Получите производный класс, который должен имитировать бильярдный стол с шестью лузами (в виде черных кружков, как показано на рис. 21.1). Когда шар попадает в лузу (достигает черного кружка), вы должны получать очко. Чтобы контролировать попадание шара в лузу, проверяйте координаты (x, y) шара. Если шар будет находиться от центра лузы на расстоянии меньшим, чем ее радиус, шар будет считаться попавшим в лузу. В этой программе радиус лузы 15, а радиус шара 10 пикселей.

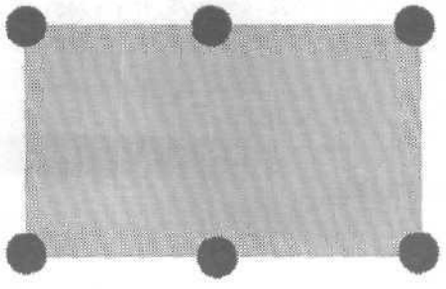


Рис. *21.1,* *Бильярдный стол*

- Ниже приведен код программы c7pool.cpp:

```
#include "franca.h"
tfinclude"math.h"
#include <stdlib.h>
const int ballradius=10, poolradius=15;
const int poolx=400, pooly=200;
class pool:public Stage
{
protected:
   Square table;
   Circle ball;
   float xb, yb;              // Координаты шара
   float incx, incy;          // Смещение
   float speed;
public:
   pool();
   void shoot(float angle);
};
pool::pool()
{
   float angle;
   speed=1.5;
   table.resize(poolx, pooly);
   table.place(320, 240);
   table.color(2);
   ball.color(0, 0);
   ball.resize(ballradius*2);
   table.show();
   insert(table);
   insert(ball);
   xb=rand()%(poolx-ballradius)+320-poolx/2.+ballradius;
   yb=randC)%(pooly-ballradius)+240-pooly/2.+ballradius;
   ball.place(xb,yb);
   ball.show();
}
void pool::shoot(float angle)
{
   angle=3.14159*angle/180.;
```

```
    incx=cos(angle)*speed;
    incy=sin(angle)*speed;
    Clock cuckoo, timer;
    for(;cuckoo, time()<30;)
    {
      if((xb<=(320-poolx/2.+ballradius))||
        (xb>=(320+poolx/2.-ball1radius))) incx=-incx;
      if((yb<=(240- pooly/2.+ballradius))||
        (yb>=(240.+pooly/2.-ballradius))) incy=-incy;
      xb=xb+i ncx;
      yb=yb+incy;
      ball.place(xb, yb);
      table.show();
      ball.show();
      timer.watch(.033);
      timer.reset();
    }
}
void mainprog()
{
    pool billiard;
    float angle;
    for(;yesno("Take a shot?");)
    {
      angle=ask("input the angle:");
      billiard.shoot(angle);
    }
}
```

Что нового мы узнали?

В этом уроке мы научились

- Использовать символьные массивы для отображения информации в проекте торгового терминала.
- Использовать массивы для обработки нескольких экранных объектов.

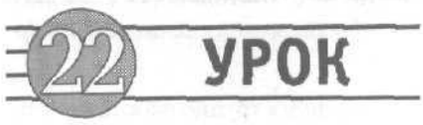
Часть VIII

Ввод и вывод данных

При реальном программировании у вас уже не будет экранных объектов, которые доступны только в программном обеспечении книги. Вам придется иметь дело со стандартным текстовым вводом/выводом, рассчитанным на клавиатуру и дисплей. В части VIII вы познакомитесь с вводом/выводом данных через стандартные потоки ввода/вывода C++. Вы также научитесь форматировать информацию ввода/вывода.

Кроме того, в реальной работе вам вряд ли захочется вводить одни и те же данные при каждом включении компьютера. Их гораздо удобнее хранить в файле на диске и при необходимости просматривать или изменять. В части VIII вы научитесь использовать файлы.

И наконец, в рамках дальнейшего развития ваших навыков создания приложений мы продолжим совершенствование торгового терминала. Наиболее существенные изменения коснутся способа хранения данных: каталог продуктов будет храниться в файле на диске.



Заголовочный файл franca.h

- Поток ввода
- Поток вывода
- Форматирование

В реальной жизни нет заголовочного файла franca.h, равно как и классов ScreenObj, athlete, Clock, а также большинства объектов других типов, с которыми мы сталкивались при изучении C++ по этой книге. Теперь, когда вы постепенно становитесь настоящим программистом, нужно учиться работать самостоятельно, без библиотек классов, разработанных специально для того, чтобы вы могли сделать свои первые шаги.

Печально, но начинающему программисту пока по-прежнему трудно работать с графическим интерфейсом Windows[1]. Поэтому мы будем изучать более скромный текстовый интерфейс, но так, чтобы владеть им достаточно уверенно. На самом деле основное различие между тем, чем мы занимались до сих пор, и тем, чем вам придется заниматься в реальной жизни, состоит в способе общения с компьютером.

Заранее определенные классы помогли нам разработать программы, создававшие на экране картинки и анимацию. Благодаря графическому интерфейсу взаимодействие пользователя с компьютером стало чуть более доверительным, В традиционном C++ интерфейс ограничен выводом данных на экран и считыванием их с клавиатуры.

На этом уроке мы сконцентрируемся на изучении ввода текстовых данных с клавиатуры, выводе их на экран и в файлы, а предварительно созданные библиотеки классов нам больше помогать не будут.



СОВЕТ Очевидно, что вы по-прежнему всегда можете воспользоваться классами, объявленными в заголовочном файле franca.h. Я также рекомендую вам продолжить изучение C++ и, в частности, программирование под Windows. Тогда пользователи ваших программ смогут взаимодействовать с ними через графический интерфейс.



ПРИМЕЧАНИЕ Кажется удивительным, но в C++ нет специальных инструкций ввода/вывода. Фактически, выводимые данные посылаются в специальные, предназначенные для этого объекты, а вводимые данные извлекаются из других таких же объектов.

[1] Поскольку создание элементов графического интерфейса Windows во всех современных компиляторах практически полностью автоматизировано, это не слишком сложно даже для начинающего программиста. Основные трудности вызывают другие аспекты программирования под Windows, в частности необходимость поддержки многопоточности и многозадачности. — *Примеч. ред.*

Реальное программирование на С++

Реальному программированию лучше всего учиться на примерах. Далее мы рассмотрим очень простую программу, в которой (как и во всех остальных наших программах) теперь останутся только стандартные для С++ заголовочные файлы. Эта программа запрашивает ваше имя и посылает вам в ответ сообщение. Выполняя эту программу, обратите внимание на первое различие между тем, что вы делали до сих пор, и тем, что вы делаете сейчас, — проектов больше нет!



ПРИМЕЧАНИЕ Проекты вам вообще больше не понадобятся. Вместо того чтобы открывать проект, удалять предыдущий файл и включать в него новый, все, что вы теперь должны делать, — это открывать файл с вашей программой и запускать его.

Теперь вы можете набрать или загрузить с прилагаемой дискеты любую программу, а затем просто запустить ее!

```
#include <iostream.h>
#include <iomanip.h>
void main()                    //c8cngrat.cpp
{
   char yourname[30];
   cout<<"Hello!"<<end<<"What is your name?";
   cin>yourname;
   cout<<endl;
   cout<<"Congratulations, "<<yourname;
   cout<<", you are now a programmer!";
}
```

В приведенном выше фрагменте кода ясно видны некоторые различия с теми упражнениями, которые мы выполняли до сих пор:

- Здесь нет заголовочного файла franca.h и, следовательно, гимнастов, бегунов, экранных объектов, часов и т. д. Кроме того, здесь нет некоторых функций, например ask(), yesno() и т. д.
- Большей частью вы будете пользоваться заголовочными файлами iostream.h и iomanip.h. Возможно, вам понадобятся и другие заголовочные файлы.



ВНИМАНИЕ Не используйте заголовочный файл franca.h вместе с заголовочным файлом iostream.h — это может привести к непредсказуемым последствиям!

- Здесь нет основной функции void mainprog(). Она теперь называется main().
- Здесь нет проектов. Программу нужно просто набрать или загрузить с дискеты, а затем запустить.



ВНИМАНИЕ Убедитесь в том, что у вас не осталось открытых проектов. Если какой-то из них открыт, закройте его.

- После запуска программ становится недоступным графический интерфейс. (Конечно, ведь вы больше не ребенок.)

Запустите эту программу и посмотрите, как она работает. Попробуйте вместо одного имени набрать имя и фамилию. Что, получается?

Проекты

Ниже перечислены некоторые рекомендации, которые помогут вам быстрее адаптироваться к новой ситуации:

- Убедитесь, что у вас нет ни одного открытого проекта. Если есть, закройте его.
- Чтобы открыть нужную программу, используйте команду File | Open главного меню.
- Для запуска программы выполните ту же процедуру, что и для запуска проекта.
- Компиляторы компании Microsoft для запуска программы создают стандартный проект. Убедитесь, что после запуска программы вы закрыли проект или среду разработки приложений.

Потоки ввода/вывода C++

Управлять вводом/выводом в C++ можно двумя способами: с помощью функций или с помощью объектов. Фактически, функции ввода/вывода пришли в C++ из языка C, «прародителя» C++.

Объекты ввода/вывода были специально разработаны в C++, и работать с ними гораздо проще и эффективнее. Для вывода данных ваша программа направляет данные в поток. В этом суть системы вывода информации в C++. Все данные, предназначенные для вывода, передаются в специальный объект (cout), который и отображает их на экране.

На самом деле все очень просто — если у вас есть целая переменная number, то вы можете написать ее значение на экране с помощью следующей инструкции:

```
cout<<number;
```

Аналогично можно вызвать другой объект (cin), который будет собирать данные, набираемые на клавиатуре. Затем данные извлекаются из этого объекта и передаются переменным программы. В этом случае ввести значение переменной number можно с помощью следующей инструкции:

```
cin>>number;
```



ПРИМЕЧАНИЕ Имена cin и cout произошли от слов C input (ввод C) и C output (вывод C),

Как вы уже заметили, операции ввода и вывода реализуются в C++ с помощью специальных операторов << и >>. Они напоминают нам, что при выводе поток направлен от переменной к объекту (cout<<number), а при вводе — от объекта к переменной (cin>>number).

Синтаксис очень напоминает тот, который мы использовали в объектах Cin и Cout из заголовочного файла franca.h.



ПРИМЕЧАНИЕ Начинающих программистов часто смущают термины *ввод* и *вывод*. Не забывайте, что они относятся к компьютеру. Для него все, что вы набираете на клавиатуре, — это *ввод, а* все, что появляется на экране, — это, соответственно, *вывод*.

Заголовочный файл iostream.h

Чтобы использовать потоковый ввод и вывод, вместо файла franca.h в программу необходимо включить новый заголовочный файл iostream.h. Это делается следующим образом:

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

Теперь вместо двойных кавычек заголовочный файл заключен в угловые скобки. Именно с помощью угловых скобок в программу включаются файлы из каталогов компилятора.

В заголовочном файле iostream.h находятся объявления и определения классов для объектов cin и cout.

Потоковый ввод/вывод обрабатывает все встроенные типы данных C++. Непосредственно могут использоваться следующие типы данных:

- int
- char
- float
- double

Что касается массивов, то для потокового ввода/вывода доступны только элементы массива и только если они относятся к встроенному типу данных. То же самое относится к структурам. Доступны только поля структуры и только если они относятся к встроенному типу данных.

Имеется, однако, одно замечательное исключение: оканчивающиеся нулем строки хотя и являются частным случаем массивов, но доступны для потокового ввода/вывода. Например:

```
char name[] = "Alfred E. Newman";
...
cout<<name;
```



ПРИМЕЧАНИЕ После того как заголовочный файл iostream.h включается в программу, автоматически объявляются и становятся доступными объекты cin и cout.

Потоковый вывод

Синтаксис потокового вывода строится из объекта cout, оператора << и переменной (или константы):

```
cout << идентификатор_переменной;
```

Например:

```
cout<<number;
```

Если выводятся значения нескольких переменных, то перед каждой из них должен стоять свой оператор вывода (<<):

```
cout<<"The result is:"<<number<<" and that is final";
```

При выводе информации на экран пробелы между значениями сами по себе не появятся. Например:

```
int number=32;
char name[]="Sonny Bonds";
cout<<name<<number;
```

После выполнения этих трех инструкций на экране появится следующее:

```
Sonny Bonds32
```

Две инструкции вывода также не приведут к выводу данных в две строки. Другими словами, в предыдущем примере результат будет тем же, даже если выводить на экран данные с помощью двух инструкций:

```
cout<<name;
cout<<number;
```

Пробел

Если требуется вставить пробел между значениями, то проще всего вывести его специально:

```
cout<<name<<" "<<number;
```

Можно использовать и более подробное описание каждого значения:

```
cout<<"Customer Name: "<<name<<" code is: "<<number;
```

Новая строка

Имеется возможность выводить данные с новой строки. Для этого существуют два способа: вывести в поток управляющий символ \n (новая строка) или манипулятор **endl**.

Создание новой строки с помощью управляющего символа:

```
cout<<"Customer Name: "<<name<<\n<<" code is: "<<number;
```

Создание новой строки с помощью манипулятора:

```
cout<<"Customer Name: "<<name<<endl<<" code is: "<<number;
```

Оба варианта приведут к появлению на экране одинаковой информации:

```
Customer Name: Sonny Bonds
  code is: 32
```

Потоковый ввод

Синтаксис потокового ввода строится из объекта **cin**, оператора **>>** и переменной, которой присваивается вводимое значение:

```
cin>>имя_переменной;
```

Следующая инструкция считывает значение с клавиатуры и сохраняет его в переменной i:

```
cin>>i;
```

С помощью одной инструкции ввода можно присвоить значения более чем одной переменной, например:

```
cin>>i>>j>>k;
```

Вы можете поэкспериментировать со следующей программой, в которой сначала запрашивается ввод трех значений, а затем эти значения выводятся на экран:

```
#include<iostream.h>
#include<iomanip.h>
void main()
```

продолжение

```
{
   int i, j, k;
   cout<<"Enter three values:\n";
   cin>>i>>j>>k;
   cout<<endl<<"Your values are:"<<\n;
   cout<<i<<" "<<j<<" "<<k;
}
```

Запустите эту программу и убедитесь, что введенные вами значения выводятся правильно.

Вы можете перейти на новую строку на экране, включив в поток либо манипулятор ввода/вывода endl, либо управляющий символ \n.

Когда управляющий символ будет найден в символьной строке или в потоке вывода, он будет интерпретирован как инструкция перехода на новую строку. Чтобы управляющий символ новой строки оказался в потоке вывода, его либо вставляют в другую строку (например, Enterthreevalues:\n), либо указывают отдельно (как во второй инструкции cout предыдущей программы). Управляющий символ может заключаться как в двойные, так и в одинарные кавычки. С точки зрения пользователя эффект будет одинаков. Однако двойные кавычки генерируют оканчивающуюся нулем строку (итого два символа), а одинарные кавычки — одиночный символ.

При вводе чисел в предыдущей программе обратите внимание на следующее:

- Между числами можно ввести один или несколько пробелов.
- После ввода каждого числа можно нажимать клавишу Enter.
- Внутри чисел (между цифрами) нельзя использовать пробелы.
- Между числами нельзя использовать другие знаки (например, запятые).

Ввод/вывод массивов

Допустим, нам нужно обработать числовой массив. Например, мы хотим просмотреть все элементы массива, чтобы определить их значения. Для этого можно использовать функцию show().

```
void show(int array[], int from, int to)
{
   cout<<endl<<"Contents of array: "<<endl<<"Index"<<"Value";
   for (int k=from; k<=to; k++)
   cout<<endl<<k<<"    "<<array[k];
}
```

У этой функции три параметра: сам массив, а также индексы начала и конца просмотра. Не забывайте, что размер массива задавать не обязательно.

Для ввода значений в массив можно использовать функцию readarray():

```
void readarray(int array[], int from, int to)
{
   for (int i=from;  i<=to;  i++)
   {
      cout<<endl<<"Please input element insdex "<<i<<: ;
      cin>>array[ i];
   }
}
```

Эта функция показывает, насколько удобно каждый раз сообщать пользователю, ввод какой величины запрашивает у него компьютер. Не *забывайте*, что вашими программами, скорее всего, будут пользоваться другие. Чем больше информации они получают о том, что требуется от них компьютеру, тем меньше ошибок будут допускать. С другой стороны, старайтесь не перегружать свои функции лишними сообщениями. Если ваша программа предназначена стать частью другого программного обеспечения, это приведет к заполнению экрана бесполезной информацией.

Вы можете проверить описанные функции с помощью программы c8show.cpp:

```
void main()            //c8show.cpp
{
   int number[10];
   readarray(number, 0, 9);
   show(number, 0, 9);
}
```

Форматирование

Большую часть своего времени профессиональные программисты тратят на получение данных и составление отчетов. Настоящие отчеты должны хорошо выглядеть. Числа и слова должны располагаться в надлежащих местах, а числа вдобавок должны быть правильно выровнены.

Это и называется *форматированием* (formatting). Как вводимые, так и выводимые данные должны быть хорошо отформатированы, чтобы люди могли легко воспринимать информацию.

Простейшее форматирование состоит в своевременных переходах на новую строку и добавлении пробелов, однако только этого не достаточно. Чтобы правильно форматировать свои отчеты, вы должны знать еще несколько приемов.

Точность значений с плавающей точкой

Когда вы выводите значения с плавающей точкой, вам может не понравиться, как они выглядят. Например:

```
#include<iomanip.h>
void main()
{
   float price, tax=6.75, total;
   cout<< "Enter the price:";
   cin>>price;
   total=price+tax*price/100;
   cout<<endl<< "Please pay: $"<<total;
}
```

Хотя данный пример и представляет простейшую программу для вычисления цены с торговой наценкой, результаты вряд ли вас удовлетворят. Если вы введете значение 10 в качестве начальной стоимости, то получите следующий результат:

```
Please pay: $10,675
```

Но значения в половину цента просто не бывает. Достаточно было бы двух значащих цифр после десятичной точки. А как этого добиться? Далее на этом уроке вы научитесь правильно округлять числа с плавающей точкой.

Выравнивание полей

Другим важным аспектом ввода/вывода является выравнивание полей. Рассмотрим предыдущую программу для считывания массива и вывода его элементов на экран. Если элементы массива будут разными, в результате выполнения программы на экране вы увидите следующее:

```
Contents of the array:
Index Value
0   1
1   23456
2   23
3   -4567
4   3
5   23
6   -21005
7   32
8   21
9   6789
```

Такой вывод не может понравиться, поскольку элементы массива выровнены плохо. Они должны не только начинаться, но заканчиваться на одной вертикальной линии. Чтобы добиться этого, нужно указать, чтобы все числа занимали при выводе полосу заданной ширины,

Манипуляторы ввода/вывода

Заголовочный файл iomanip.h определяет несколько манипуляторов, которые могут быть включены в поток ввода/вывода для форматирования данных. До сих пор мы рассматривали только манипулятор endl, но есть и другие (табл. 22.1).

Таблица 22.1. *Манипуляторы заголовочного файла iomanip.h*

Манипулятор	Назначение
endl	Начало новой строки
ends	Вставка символа null в поток вывода
flush	Очистка потока
setiosflags(long flag)	Установка флагов ввода/вывода
resetiosflags(long flag)	Очистка флагов ввода/вывода
setfill(char fillchar)	Установка символа-заполнителя
setprecision(int places)	Установка точности
setw(int width)	Установка полной ширины поля

Манипуляторы setiosflags, resetiosflags, setfill и setprecision сохраняют свое действие до тех пор, пока не будут переопределены. Например, если в качестве символа-заполнителя определить точку, то он будет оставаться таковым, пока не определен другой символ-заполнитель. В то же время манипулятор setw действует однократно.

Манипуляторы включаются в поток тем же способом, что и обычные переменные. Помните, как мы писали манипулятор endl? Например, чтобы должным образом выводить на экран элементы массива, нужно следующим образом модифицировать функцию show():

```
void show(int array[], int from, int to)
{
   cout<<endl<< "Contents of array: "<<endl<< "Index"<<" Value";
   for (int k=from; k<=to; k++)
   cout<<endl<<setw(3)<<k<< "     "<<setw(8)<<array[k];
}
```

Различия в компиляторах

Компиляторы компании Microsoft не позволяют сразу изменить заданное ранее выравнивание установкой флагов в положение ios::left или ios::right. Пусть, например, установлено выравнивание влево:

```
setiosflags(ios::left)
```

Тогда, чтобы изменить выравнивание, нужно сначала отменить предыдущую установку:

```
resetiosflags(ios::left)
```

Компиляторы компании Borland позволяют изменять выравнивание без отмены предыдущей установки.

Манипулятор setw

Предыщущая модификация функции show() иллюстрирует достоинства манипулятора setw. В этом случае каждое значение индекса k будет занимать ровно три позиции, вне зависимости от того, сколько на самом деле требуется для него места. Точно так же каждое значение элемента массива будет занимать ровно восемь позиций. Ширину необходимо предусмотреть достаточной для выводимого значения. Если ширина окажется недостаточной, компьютер использует дополнительные позиции, чтобы вывести ваше число с максимальной точностью (при этом формат будет нарушен). Манипулятором по умолчанию является setw(0), который задает минимальное число позиций для исчерпывающего представления значения.

Манипулятор setprecision

При выводе чисел с плавающей точкой может потребоваться ограничивать их точность. Например, при обработке денежных значений часто требуется, чтобы результат содержал только два десятичных разряда. Для этой цели можно воспользоваться манипулятором setprecision. После определения числа десятичных разрядов это определение остается в силе до появления другого манипулятора setprecision. Например, чтобы не выводить цену с более чем двумя десятичными разрядами, необходимо сделать следующее:

```
float price;
cout<<setprecision(2);
...
cout<<price;
```

Пока вы вновь не воспользуетесь этим манипулятором, любая переменная с плавающей точкой, направляемая в объект cout, будет выводиться только с двумя десятичными разрядами. Несомненно, этот манипулятор позволит вам решить много других проблем.

Если ваше число с плавающей точкой содержит менее двух десятичных разрядов, то оно так и будет выведено. Более того, десятичная точка может быть даже опущена. Например, пусть имеются следующие инструкции:

```
float x[3]={12.35, 10., 5.};
cout<<steprecisiоn(2);
for(int i=0, i<3, i++);
cout<<end<<setw(8)<<x[i];
```

Тогда после их выполнения экран будет выглядеть так:

```
12.35
10
5
```

Такое форматирование данных, возможно, и не подойдет для вашего отчета. Для установки положения десятичной точки необходимо воспользоваться манипулятором setiosflags.

Другая проблема, с которой вы можете столкнуться при установке ширины и/или точности, состоит в том, что компьютер может выводить значения в экспоненциальной (научной) нотации. Если вам это не подходит, то снова воспользуйтесь манипулятором setiosflags.

Манипулятор setfill

Когда вы (с помощью манипулятора setw) определили ширину поля, а в выводимом значении меньше позиций, чем определено, оставшиеся позиции будут заняты пробелами. В этом случае символом-заполнителем становится пробел. Однако вместо него в этом качестве может быть и любой другой символ. Достигается это с помощью манипулятора setfill. Как только символ заполнитель определен, он будет использоваться вплоть до следующего манипулятора setfill. Чтобы опять вернуться к пробелу в качестве символа-заполнителя, необходимо записать манипулятор setfill(' ').

Например, при выдаче чека с выраженными в долларах значениями в качестве символа-заполнителя принято использовать символ *:

```
cout<<"US$"<<setfill(*)<<value;
```

В других ситуациях, возможно, больше подойдут нули, точки или что-нибудь еще.

Манипуляторы setiosflags и resetiosflags

Манипуляторы setiosftags и resetiosflags в операциях ввода/вывода используются для установки флагов. Эти флаги перечислены в табл. 22.2.

Таблица 22.2. Флаги заголовочного файла iomanip.h

Флаг	Назначение
skipws	Игнорирование разделителей в потоке ввода
left	Выравнивание влево
right	Выравнивание вправо
showpoint	Отображение десятичных позиций и точки
scientific	Научный (экспоненциальный) формат
fixed	Формат с фиксированной точкой

Флаг устанавливается с помощью манипулятора setiosflags, аргумент которого состоит из символов ios:: и устанавливаемого флага. Чтобы отменить действие флага, используется манипулятор resetiosflags с тем же аргументом. Например, следующая инструкция отменяет действие флага skipws и позволяет вам считывать разделители (обычно пробелы) в потоке ввода:

```
cin>>resetiosflags(ios::skipws);
```

Флаг skipws

Флаг skipws установлен по умолчанию. При этом невозможно считывать пробелы, которые обычно используются в качестве разделителей между значениями. Например, если считывать строку в массив символов, то поток ввода закончится на первом встретившемся в строке пробеле.

Вы можете попробовать запустить следующую программу:

```
#include <iomanip.h>
void main()
{
   char mark[]="_____+";
   char name[30];
   cin>>name;
   cout<<name;
}
```

Если вводить, например, строку Sonny Bonds, в массиве окажется только слово Sonny, поскольку пробел будет воспринят как конец строки.

В следующей инструкции мы отменяем действие флага skipws:

```
cin>>resetiosflags(ios::skipws);
```

Но и после отмены действия флага skipws мы снова получаем тот же самый результат! Как же тогда считывать разделители (пробелы, табуляции и т. д.)?

Как уже упоминалось, C++ автоматически считывает символьные массивы как строки. Чтобы разделители тоже вошли в массив, нужно не только отменить действия флага skipws, но и считывать все символы по одному. Рассмотрим следующую программу:

```
ftinclude <iomanip.h>
void main()
{                    //c8skipws.cpp
   const char enter=10;
   char mark[]="----+";
   charname[30];
   cin>>resetiosflags(ios::skipws);
   cout<<mark<<mark<<mark<<endl;
   for(int i=0; i<29; i++)
   {
      cin>>name[i];
      if(name[i]==enter) break;
   }
   name[i]=0;]
   cout<<endl<<setw(5)<<i<<"characters read";
   cout<<endl<<name;
}
```

В этой программе отменяется действие флага skipws и вводимая строка считывается по одному символу. Чтобы не считывать все 30 символов, программа проверяет, не нажимал ли пользователь клавишу Enter. Для этого код символа сравнивается с кодом клавиши Enter (определенном в константе, численное значение которой равно 10).

Если вы не вызовете манипулятор resetiosflags, строка все равно будет считана, но разделители при этом будут проигнорированы. Более того, код клавиши Enter тоже не будет обнаружен, и вам на самом деле придется считать все 30 «непустых» символов.

Массив символов mark используется исключительно для того, чтобы помочь вам следить за тем, сколько символов напечатано.

Выравнивание значений

Когда для значений требуется меньше позиций, чем доступно, можно выровнять их влево или вправо, установив соответствующий флаг. По умолчанию выравни-

вание производится вправо (именно так, скорее всего, должны отображаться числа), но это не совсем то, что нужно для строк символов.

Рассмотрим следующую программу:

```
#include <iomanip.h>
void main()
{
   charname[5][20];
   int code[5];
   for(int i=0; i<=4; i++);
   {
      cout<<endl<<"Enter a name:";
      cin>>name[i];
      cout<<endl<<"Enter a code:";
      cin>>code[i];
   }
   for(i=0; i<=4; i++)
   {
      cout<<endl<<setw(20)<<name[i]<<setw(6)<<code[i];
   }
}
```

Эта программа считывает последовательность имен и чисел с клавиатуры и затем отображает их на экране. Результат может выглядеть следующим образом:

```
Clarice        12
    Liz      4532
  Lucia       435
 Marcos        21
Claudio        43
```

Выравнивание текста и чисел

Приведенное выше форматирование нельзя назвать удачным. Лучше выравнивать имена влево, а числа вправо. Новая программа может быть следующей:

```
#include <iomanip.h>
void main()
{
   charname[5][20];
   int code[5];
```

```
    for(int i=0; i<=4; i++);
    {
       cout<<endl<<"Enter a name:";
       cin>>name[i];
       cout<<endl<<"Enter a code:";
       cin>>code[i];
    }
    for(i=0; i<=4; i++)
    {
       cout<<endi«setiosflags(ios::left)<<setw(20)<<name[i]
              <<resetiosflags(ios::left)
              <<setiosflags(ios::right)<<setw(6)<<code[i];
    }
}
```

Теперь, после выполнения программы, мы увидим на экране другую картину:

```
Clarice      12
 Liz       4532
Lucia       435
Marcos       21
Claudio      43
```



ПРИМЕЧАНИЕ Действие флага выравнивания влево было отменено, чтобы программа оставалась работоспособной в среде компиляторов Microsoft.

Флаг showpoint

Флаг showpoint при выводе чисел с плавающей точкой ведет к отображению десятичной точки и завершающих нулей. При этом число занимает ровно столько позиций, сколько задано манипулятором setprecision. Без флага showpoint десятичная точка и завершающие нули могут не появиться (если реальное число с плавающей точкой занимает меньше позиций, чем задано манипулятором setprecision).

Флаг scientific

Флаг scientific ведет к отображению чисел с плавающей точкой в экспоненциальном (научном) формате.

Флаг fixed

Действие флага fixed отменяет действие флага scientific и ведет к отображению чисел с плавающей точкой в формате с фиксированной точкой.

Коды символов

На уроке 20 мы познакомились с типом данных char и узнали, что для представления символов используется код ASCII — целое число в диапазоне от 0 до 255. Запоминать, какой код соответствует тому или иному символу, не обязательно, для этого можно воспользоваться следующей программой:

```
#include <iostream.h>
void main()            //c8ascii.cpp
{
// Соответствие символов и их кодов
char code, choice='y';
int  number;
while(choice=='y')
{
        cout<<endl<<"Input the code you want to know:";
        cin>>number;
        code=number;
        cout<<endl<<"This code corresponds to the charcter:"<<code;
        cout<<endl<<"Do you want to continue(y/n)?";
        cin>>choice;
    }
}
```

Эта программа запрашивает ввод целого числа, а затем выводит на экран символ, код которого вы ввели. Обратите внимание, что переменная code была объявлена как символ, в то время как переменная number — как целое. Поэтому при выводе символа компьютер понимает, что вы хотите видеть не число, а символ, код которого равен числу. Даже если значения переменных code и number одинаковы и равны 65, при выводе на экран значения переменной number вы увидите число 65, а при выводе значения переменной code — символ A.

Приведенная выше программа просто считывает целое число, копирует его в символьную переменную (code) и затем выводит эту переменную на экран.



ПРИМЕЧАНИЕ Если вы хотите явно обозначить необходимость преобразования целого в символ, можете задать операцию приведения типов в инструкции присваивания: code=(char)number;. Однако это не обязательно, поскольку в данном случае компилятор выполняет операцию приведения типов автоматически.

Функции ask(), askwords() и yesno()

Функции ask(), askwordsQ и yesno(), а также объекты класса Box были очень полезны для обработки ввода/вывода на предыдущих уроках. Хотя вы не можете ис-

пользовать графический интерфейс Windows со стандартными потоками ввода/вывода, я все же предоставлю вам возможность вернуться (с некоторыми ограничениями) к нашим функциям и информационным рамкам.

Заголовочный файл nofranca.h

Заголовочный файл nofranca.h позволяет вам снова использовать в ваших программах перечисленные выше функции и информационные рамки. Все, что вам нужно сделать, — это включить в программу следующую директиву:

```
#include "nofranca.h"
```

Включение в программу этого нового заголовочного файла предотвратит включение туда файла franca.h. Любые вызовы этих функций и использование информационных рамок будет интерпретировано согласно новому заголовочному файлу и отражено в интерфейсе C++,

Полезно изучить этот файл. Тогда вы сможете понять работу программы в целом. Обратите внимание на директивы, в которых проверяются файлы FRANCA_H и _CANVAS_H. Они предотвращают загрузку файла franca.h, если загружен файл nofranca.h.

Код файла nofranca.h

Ниже приведен полный код файла nofranca.h, который можно найти на прилагаемой дискете;

```
ftifndef    NOFRANCA_H
ffdefineNOFRANCA_H
#define FRANCA_H
#define _CANVAS_H
#include<iostream.h>
#include <string.h>
#include<stdlib.h>
#include <iomanip.h>
float ask(char question[])
{
   floatanswer;
   cout<<endl<<"Asking:"<<question;
   cin>>answer;
   return answer;
}
void askwords(char sentence[], int size, char question[])
{
   cout<<endl<<"Asking(sentence):"<<question;
   cin>>sentence;
}
```

продолжение

```
int yesno(char sentence[])
{
   char answer;
   cout<<endl<<"yes or no: "<<question;
   cout<<endl<<"Please enter y or n: ";
   cin>>answer;
   if(answer=='y) return 1;
   return 0;
}
class Box
{
char title[40];
   char message[40];
public:
   Box();
   Box(char alabel[]);
   void say(float);
   void say(char msg[]);
   void label(int);
   void label(char msg[]);
   void place(int x, int y);
};
void Box::Box()
{
   strcpy(title, " ");
   strcpy(message, " ");
}
void Box:: Box(char msg[])
{
   strcpy(title,msg);
}
void Box::place(int x, int y)
{

}
void Box: :say(float value)
{
   cout<<endle<<title;
   cout<<endle<<setprecision(2)<<setiosflags(ios::showpoint);
   cout<<setiosflags(ios::fixed)<<value;
}
```

```
void Box::say(charmsg[])
{
   cout<<endle<<title;
   cout<<endle<<msg;
}
void Box::label(charmsg[])
{
   strcpy(title,msg);
}
void Box::label(int value)
{
   itoa(value, title, 10);
}
ttendif
```

Самостоятельная практика

- Измените программу c8ascii.cpp так, чтобы она считывала символ и выводила соответствующий код (число),
- Напишите программу, которая отображала бы таблицу с двумя столбцами. Первый столбец должен содержать целые от 0 до 255, а второй — символы, соответствующие кодам в первом столбце.
- Измените предыдущую программу так, чтобы все коды уместились на экране. Для этого таблица должна выводиться в виде 10 групп по 2 столбца в каждой.

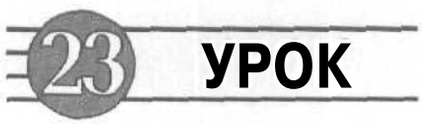
Что нового мы узнали?

В этом уроке мы научились

0 Обходиться без заголовочного файла franca.h.

0 Использовать потоковый ввод/вывод,

0 Форматировать ввод/вывод.



УРОК 23 Файлы

- Использование файлов
- Директивы препроцессора
- Создание класса для текстовых файлов
- Ввод из файла и вывод в файл
- Базовые операции с файлами

На уроке 23 **мы будем** учиться сохранять наши данные в файлах. Это будут стандартные потоковые файлы C++. Мы создадим свой собственный класс для работы с файлами. Кроме того, мы познакомимся с базовыми файловыми операциями — поиском и сравнением. **На** следующем уроке мы воспользуемся этими приемами для дальнейшего совершенствования торгового терминала.

Использование файлов

С самого начала этой книги мы сохраняли программы на жестком диске вашего компьютера, поэтому их не приходилось каждый раз набирать заново. Вообще говоря, любая хранящаяся на диске информация называется файлом. Здесь вы узнаете, как сохранять в файлах данные ваших программ, чтобы их можно было затем считывать или переносить на другой компьютер.

Работа с файлами аналогична работе с дисплеем или клавиатурой. Конечно, поскольку у вас уже имеется некоторый опыт работы с программными файлами, то вы знаете, что некоторые незначительные отличия все же имеют место:

- Файл идентифицируется по своему имени на диске.
- Можно использовать один и тот же файл как для записи информации (вывод из компьютера), так и для считывания информации (ввод в компьютер).

Поскольку файлу приходится давать имя, файловый объект невозможно объявить компилятору заранее. Объекты потоков ввода/вывода были изначально объявлены с идентификаторами cin и cout, поэтому специальные имена им можно было не назначать. Ситуация с файлами иная. Вам может потребоваться несколько файлов для ввода (раньше это был один объект cin) и для вывода (раньше это был один объект cout). Следовательно, сначала необходимо объявить свои файлы в качестве объектов класса fstream и только потом дать им соответствующие имена.

Пользоваться функциями-членами класса fstream можно только после объявления объектов этого класса. Когда объекты объявлены, задать поток ввода и/или поток вывода для файлов можно точно так же, как это делалось для объектов cin и cout. При этом доступны те же манипуляторы и флаги!

Например:

```
ffinclude<fstream.h>
void main()
{
   fstream myfile;
   myfile.open("list.txt", ios::out);
   myfile<<"Anything goes";
   myfile.close();
}
```

В этой программе создается файл с именем list.txt, и в него записывается строка Anything goes.



ПРИМЕЧАНИЕ Для работы с файлами в программу необходимо включить заголовочный файл fstream.h.

Вы можете запустить эту программу и в любом текстовом процессоре просмотреть содержимое файла list.txt. Можно также распечатать содержимое файла. Поскольку мы не назначили каталог для хранения файла, он будет создан в текущем каталоге.

Обработка файлов

Можно обрабатывать два вида файлов:

- Текстовые файлы.
- Бинарные файлы.

Текстовые файлы организованы в строки, каждая из которых помечена символом *конца строки* (endl). При этом все данные хранятся в удобочитаемом виде. Поля данных разделены пробелами, поэтому содержимое файла можно легко понять. Напротив, бинарные файлы не организованы в строки. Одна часть информации непрерывно переходит в другую, а числа хранятся так же, как и в памяти компьютера, в бинарном формате, а не в виде привычных нам десятичных цифр. Поэтому прочесть распечатанный бинарный файл совсем не просто. В этой книге мы будем работать исключительно с текстовыми файлами.

Чтобы можно было использовать файл, его, как и любой другой объект, необходимо объявить.

Объявление файла

Поскольку файлы являются объектами класса fstream, их объявление выглядит очень просто:

```
fstream идентификатор;
```

Идентификатор — это имя, которое используется в программе для обращения к файлу, а не имя файла на диске! Соответствие между идентификатором и именем файла на диске определяется *функцией-членом* open().

После объявления файла операции над ним можно выполнять с помощью функций-членов класса fstream. Наиболее важными функциями-членами являются:

- open()
- close()
- eof()

Функция-член open()

Функция-член open() класса fstream устанавливает соответствие между файловым объектом и данными на диске. Она определяет также режим доступа к файлу: для ввода, для вывода, для добавления и т. д. Синтаксис функции open():

```
open(char имя_файла[], int режим_доступа);
```

Функция получает два аргумента: строку символов (имя_файла), определяющую имя файла на диске, и флаг (режим_доступа), определяющий режим доступа к файлу.

Имя файла

Имя файла является оканчивающейся нулем строкой. В строке можно задать полный путь к файлу, включая диск, каталог и т. д. При этом обратная косая черта \ в полном пути к файлу в аргументе функции заменяется двойной обратной косой чертой \\.

Режим доступа

Режим доступа может быть одним из следующих:

- Флаг ios::out открывает файл для вывода (запись в файл).
- Флаг ios::in открывает файл ввода (считывание из файла).
- Флаг ios::app открывает файл для добавления (запись в конец предыдущей информации).
- Флаг ios::nocreate открывает файл только в том случае, если он существует.

Например:

```
// Открытие файла list.txt, который находится на диске a
// в каталоге \franca, для ввода:
myfile.open("a:\\franca\\list.txt", ios::in);
// Открытие файла, имя которого написано в
// символьном массиве "имя_файла", для вывода:
myfile.open(имя_файла, ios::out);
```

Флаги режима доступа можно комбинировать, используя логический оператор дизъюнкции (|). Например:

```
ваш_файл.open("roster.txt", ios::nocreate|ios::in);
```

В этом случае система попытается открыть для ввода (флаг ios::in) файл roster.txt. Если этот файл в текущем каталоге не найден, новый файл создан *не будет* и открытие естественно не состоится.



ПРИМЕЧАНИЕ В компиляторах компании Microsoft при попытке открыть несуществующий файл всегда создается новый. Чтобы избежать этого, необходимо использовать флаг ios::nocreate. В компиляторах компании Borland в аналогичной ситуации (при попытке открыть несуществующий файл) новый файл не создается.

Проверка правильности открытия файла

Всегда полезно проверить, насколько правильно открылся файл. Например, файл может не открыться, если его нет в указанном месте. В компиляторах компаний Microsoft и Borland такая проверка выполняется по-разному.

При проверке правильности открытия файла в компиляторах Microsoft вызывается функция-член is_open(). Если функция возвращает ноль, значит, файл не открылся. Используется функция is_open() следующим образом:

```
ваш_файл.open("roster", ios::in|ios: :nocreate);
if(ваш_файл.is_open()==0) cout<<"Ошибка открытия файла";
```

При проверке правильности открытия файла в компиляторах Borland необходимо сравнить объект класса fstream с константой NULL. Если результатом сравнения является истина, значит, файл не открылся. Например:

```
ваш_файл.open("roster", ios::in|ios::nocreate);
if(ваш_файл==NULL) cout<<"Ошибка открытия файла";
```

Функция-член close()

Функция-член close() обеспечивает правильное закрытие файла и сохранение данных на диске. Иногда данные попадают на диск не сразу, и при неправильном закрытии файла могут быть утеряны. Эта функция без параметров.

Функция-член eof()

Функция-член eof() очень полезна при открытии файлов для ввода. Она возвращает 1, если в процессе считывания данных вы переходите за границу файла, которая отмечается символом конца файла (символ end-of-file или просто eof). Удобство функции eof() определяется тем, что в большинстве случаев объем данных в файле заранее не известен. Когда же нужно прекращать считывание? Когда будет достигнут символ eof.

Первый пример считывания и записи информации

Файл c8persnl.txt прилагаемой дискеты содержит информацию о сотрудниках некоего предприятия. Вы можете использовать любой текстовый процессор, чтобы прочесть и распечатать этот файл. Каждая строка в нем содержит идентификационный номер, имя и фамилию сотрудника, а также число с плавающей точкой, показывающее его почасовую оплату.

Содержимое файла выглядит следующим образом:

```
1  Clark Kent       20.00
2  Alfred Newman     6.00
3  Oliver Twist     15.00
4  Huck Finn        17.00
7  James Bond       32.00
10 Saint Nick       22.00
```

Обратите внимание, что идентификационные номера не образуют полную последовательность (некоторые из них, например 5, 6, 8 и 9, пропущены), что типично для большинства файлов. В реальной ситуации вы не будете знать заранее, данные о скольких сотрудниках вы будете считывать. Вот почему необходимо проверять наличие символа *конца файла*.

Попробуем тоже написать аналогичную программу. Хотя это достаточно простая программа, она требует при форматировании определенного внимания. Идея проста:

1. Объявить файл и переменные.
2. Считывать информацию, пока не будет достигнут конец файла.
3. Вывести идентификационный номер, имя, фамилию и почасовую оплату каждого сотрудника.

Начать составление программы можно с комментариев, которыми станут описанные выше этапы, а потом вставить соответствующий программный код:

```
#include <fstream.h>          //c8wage.cpp
#include <iomanip.h>
void main()
{
   // Объявить файл и переменные:
   char filename[]="c8persnl.txt";
    int id;
   char fname[20], lname[20];
    float wage;
   fstream employees;
   employees.open(filname, ios::in | ios::nocreate);
```

продолжение

```
   // Считывать информацию, пока не будет достигнут конец файла:
   for(;;)
   {
      employees>>id;
      if(employees.eof())break;
      employees>>fname>>lname>>wage;
   // Вывести идентификационный номер, имя, фамилию
   // и почасовую оплату каждого сотрудника:
      cout<<endl;
      cout<<setw(5)<<setiosflags(ios::right)<<id
         <<setw(20)<<setiosflags(ios::left)<<fname
         <<setw(20)<<lname
         <<setw(6)<<setprecision(2)<<setiosflags(ios::fixed)
         <<resetiosflags(ios::left)
         <<setiosflags(ios::showpoint)<<wage;
   }
   char choice;
   cout<<endle<<"Enter any character to finish";
   cin>>choice;
}
```

Обратите внимание на следующие аспекты приведенной программы:

- Мы инициализировали символьный массив filename именем файла. Это сработает только в том случае, если файл находится в том же каталоге, что и программа. В противном случае необходимо указать полный путь.
- Мы не стали проверять, насколько успешно открылся файл.
- Мы проверили наличие символа конца файла. Важно ввести такую проверку сразу после начала считывания данных. Напоминаю, что символ конца файла является последним символом файла. Поэтому если вы вставите проверку соответствующего условия в начало цикла (например, используя инструкцию while(!employees.eof()...)), то на последней итерации вы будете работать с неправильными данными.
- Мы отформатировали вывод, что является, вообще говоря, трудной задачей, поскольку данные поочередно нужно выравнивать по левому (имена) и по правому (числа) краю. Компиляторы Microsoft требуют, чтобы вы отменили действие флага выравнивания влево, вместо того чтобы просто указать нужный флаг. Кроме того, нужно было установить специальный флаг, чтобы компиляторы Microsoft показывали положение десятичной точки и десятичные нули.
- Последние три строки не обязательны. Некоторые компиляторы очищают экран вскоре после завершения программы. Когда это происходит, у вас едва хватает времени просмотреть результаты. Выводя на экран запрос о вводе информации, вы заставляете компьютер не очищать экран.

Кроме этого, может оказаться полезным добавить в программу еще несколько элементов:

- Проверку правильности открытия файла.
- Форматирование имени и фамилии.

Чтобы проверить правильность открытия файла, необходимо включить в программу инструкцию if, содержание которой у разных компиляторов будет разное. В компиляторе Borland эта инструкция выглядит следующим образом:

```
if(employee==NULL) cout<<"Error: file does not open";
else ...
```

В компиляторе Microsoft нужно использовать следующую инструкцию:

```
if(employee.is_open()==0) cout<<"Error: file does not open";
else ...
```

Отметьте, что различие касается только условия инструкции if.

Вас не раздражает тот факт, что в зависимости от типа компилятора приходится менять свой код?

Фактически, одно из преимуществ использования языка программирования типа C++ состоит в том, что программы не требуют изменения при переходе от одной машины к другой или от одного компилятора к другому. Тем не менее многие компиляторы имеют незначительные несоответствия в способе выполнения некоторых процедур.

Скоро вы узнаете, как пользоваться препроцессором, чтобы получить возможность с разными компиляторами работать одинаково.

Выводимый предыдущей программой список может оказаться неудовлетворительным еще по одной причине: имя и фамилия находятся в разных столбцах. Изменить ситуацию можно, копируя имя в новый символьный массив и присоединяя к нему сначала пробел, а затем фамилию. Например:

```
char full_name[40];
...
strcpy(full_name, fname);
strcpy(full_name, " ");
strcpy(full_name, lname);
```

Эта последовательность инструкций приведет к тому, что символьный массив full_name будет содержать полное имя сотрудника, которое можно будет распечатать вместо отдельных имени и фамилии. Окончательная реализация программы приводится в следующем примере.

Второй пример считывания и записи информации

Ниже показана новая версия предыдущей программы. В этой новой версии для представления сотрудника используется объект, а форматирование осуществляется функцией-членом display(). Для иллюстрации альтернативного подхода объект строится на базе структуры, а не класса.

```
#include <fstream.h>
#include <iomanip.h>
#include <string.h>
struct hired_person
{
   int id;
   char fname[20], lname[20];
   float wage;
   void display();
};
void hired_person::display()
{
   char full_name[40];
   strcpy(full_name, fname);
   strcat(full_name, " ");
   strcat(full_name, lname);
   cout<<setiosflags(ios::right)<<setw(6)<<id<<" "<<setw[40]
      <<setiosflags(ios::left)<<full_name<<setw(6)<<setprecision(2)
      <<resetiosflags(ios::left)
      <<setiosflags(ios::right)
      <<setiosflags(ios::fixed)
      <<setiosflags(ios::showpoint)<<wage;
}
```

Благодаря этой структуре основная функция программы c8wage1.cpp меняется следующим образом:

```
void main()        //c8wage1.cpp
{
   // Объявление переменных:
   char filename[]="c:\\franca\\c8persnl.txt";
   hired_person worker;
   fstream employees;
```

```
    // Открытие файла для ввода:
    employees.open(filename, ios::in);
    // Цикл просмотра всех сотрудников;
    {
        for(;;)
        {
            // Считывание данных на сотрудника:
            employees>>worker.id;
            // Проверка конца файла:
            if(employees.eof()) break;
            employees>>worker.fname>>worker.lname>>worker.wage;
            cout<<endl;
            // Вывод данных на сотрудника:
            worker.display();
        }
        cout<<endl<<endl<<"Enter any character to finish:";
        char enter;
        cin>>enter;
    }
}
```

Если функцию display() сделать функцией-членом, станет проще основная функция программы, причем не только концепция, но и реализация, поскольку изменение информации файла уже не будет сильно сказываться на программе. Однако, чтобы получить из этого максимальную выгоду, необходимо все операции над файлом тоже инкапсулировать в классе. И наконец, считывание информации тоже необходимо выполнять не в основной функции программы, а в классе.

Проверка правильности открытия файла

Некоторые программисты предпочитают укороченную форму условия проверки правильности открытия файла. То есть вместо инструкции:

```
if(employee.is_open()==0) cout<<"Error: file does not open";
```

используется инструкция:

```
if(employee.is_open()) cout<<"Error: file does not open";
```

Оба варианта ведут к одинаковому результату.

Директивы препроцессора

Познакомимся с новыми для нас директивами препроцессора:

Otfdefine

Otfifdef

Otffndef

Otfendid

Эти директивы могут пригодиться во многих случаях, однако основная наша задача в этом разделе состоит в преодолении различий в работе компиляторов.

Помните, что препроцессор обрабатывает вашу программу *до* ее компиляции. При функционировании программы директивы препроцессора не работают.

Директива ftdefine

Директива tfdefine может применяться для замены данного идентификатора данной строкой. Например:

```
#define pi 3.1416
```

При задании этой директивы препроцессор будет искать в оставшейся части программы идентификатор pi и заменять его строкой (в данном случае последовательностью цифр) 3.1416. Результат будет тем же, что и при выполнении инструкции:

```
const float pi=3.1416;
```

Однако первым способом можно достичь гораздо большего, поскольку любой идентификатор может быть заменен любой строкой, если в ней нет разделителей (пробелов, табуляций и т. д.). Например, можно написать следующие директивы:

```
#define условие1 employee.is_open()
tfdef ine условие2 employee==NULL
```

Тогда внутри программы условие проверки правильности открытия файла не будет зависеть от компилятора:

```
if(condition1) cout<<"Error";
```

Тем не менее разница в программах останется, поскольку придется использовать либо условие1, либо условие2. Хорошо бы иметь возможность выбирать, каким условием пользоваться.

Директива tfifdef

Директива ftifdef проверяет, был ли данный идентификатор определен директивой ftdefine. При этом не важно, *как* он был определен, а важно *был ли* он определен. В случае использования одного из компиляторов (Borland или Microsoft) нужно написать *одну* из следующих директив:

```
ttdefine Borland
#define Microsoft
```

Задав таким способом тот или иной компилятор, вы впоследствии можете выяснять тип своего компилятора, просто проверяя, был ли определен соответствующий идентификатор. Например:

```
ttifdef Borland
   tfdef ine condition employee==NULL
flendif
ffifdef Microsoft
   #define condition employee. is_open()==0
#end if
```

Директива ttendif

В C++ для обозначения начала и конца действия инструкции if используются фигурные скобки {}. В препроцессоре соответствующее действие начинает действовать сразу после директивы tfifdef и заканчивается после первой директивы tfendif. Поэтому в предыдущую последовательность директив были включены две директивы #endif.

Если был определен идентификатор Borland, то соответствующее условие condition для инструкции if будет определено как employee==NULL, то есть так, как и требуется при работе с компилятором Borland. Если же идентификатор Borland определен не был, то препроцессор будет продолжать проверку директив, пока не встретит директиву ttendif. Далее он проверит, был ли определен идентификатор Microsoft и т. д.

В соответствии с тем, какой компилятор был обозначен, идентификатору condition будет присвоена одна из следующих строк:

```
employee==NULL
employee, is_open()==0
```

Таким образом, в программе можно использовать инструкцию:

```
if (condition) cout<<"Error";
```

А что если ни одна из опций не была определена?

В этом случае значение не будет присвоено идентификатору condition, что приведет к ошибке в программе. Если вы этого не хотите, то можете вставить дополнительное условие в виде строки, которая никогда не будет истинна. Теперь, когда не определены оба идентификатора Borland и Microsoft, препроцессор просто движется дальше. Например:

```
ttdefine condition 0!=0
ttlfdef Borland
   #define condition employee==NULL
ftendif
```

```
ftifdef Microsoft
   #define condition employee.is_open()==0
tfendif
```



ПРИМЕЧАНИЕ Действие директивы #ifndef противоположно действию директивы #ifdef, Если идентификатор не определен, то препроцессор будет обрабатывать следующие строки. Если идентификатор определен, то препроцессор пропустит следующие директивы и сразу перейдет к директиве #endif.

Проверка правильности открытия нескольких файлов

Наша препроцессорная техника проверки правильности открытия файла имеет существенный недостаток: она проверяет только один определенный файл (employee). А если файлов несколько?

В этом случае лучше всего комбинировать директивы препроцессора с функцией, параметром которой является файл.

Функция, которую я назвал fileopen(), будет проверять, открылся ли заданный файл, и, если да, возвращать 1, а если нет — 0.

Бот ее возможная реализация:

```
#include <fstream.h>
int fileopen(fstream &datafile)
{
   int yesopen=1;
   #ifdef Microsoft
      yesopen=datafile.is_open();
   #endif
   tflfdef Borland
      yesopen=!(datafile==NULL);
   #end if
   return yesopen;
}
```

Эта функция, в частности, может быть встраиваемой.

При компилировании этой функции препроцессор проверит, какой идентификатор вы определили, Borland или Microsoft (только не определяйте их вместе). Например, если используется компилятор Borland, то код для компилятора Microsoft будет пропущен и функция примет следующий вид:

```
int yesopen=1;
   yesopen=!(datafile==NULL);
return yesopen;
```

С другой стороны, если используется компилятор Microsoft, то будет пропущен код для компилятора Borland.

Все, что вам теперь нужно сделать, — это включить функцию fileopen() в программу:

```
if(fileopen(employee)==0) cout<<"Error";
```

Код этой функции включен в заголовочный файл c8tfile.h и используется классом textfile, о котором рассказывается в следующем разделе.

Самостоятельная практика

- Модифицируйте программу c8wage1.cpp, поместив операцию *считывания* внутрь класса. Для этой цели включите в класс функцию-член read(). Эта функция должна проверять условие конца файла и возвращать 1, когда все данные успешно считаны, а ноль в противном случае.

Класс textfile для обработки файлов

Поработав с файлами в нескольких приложениях, вы, возможно, захотите повторно использовать программные фрагменты, предназначенные для обработки файлов. Для этого *можно* определить класс, позволяющий выполнять с файлами все операции, которые мы до сих пор изучали.

Все операции с файлам должны быть удалены из основной функции. Объект нового класса textfile (или производного от него) должен уметь обрабатывать файлы сам.

Если в нашем примере с платежными ведомостями был бы доступным такой класс workers (производный от класса textfile), то действие основной функции выглядело бы следующим образом:

```
{
   workers staff;   // Объявление объекта класса сотрудников
   for(;;)
   {
      if(staff.read()==0)break;
      staff.display();
   }
}
```

Обратили внимание, что нам дает новый класс?

- Не нужно открывать файл — он будет автоматически открыт конструктором при объявлении объекта.

- Не нужно даже упоминать объект fstream и вообще выполнять операции с файлами в основной функции программы.
- В основной функции не нужна информация о структуре записей (строк). Не нужно считывать каждое поле, как это было раньше. Можно использовать функцию-член read(), чтобы сразу считать всю запись.
- Функцию-член read() можно реализовать так, чтобы в качестве результата она возвращала целое значение. Например, ноль означал бы конец файла.

Конечно, формат записи может быть свой у каждого типа файлов, с которыми вы будете работать. Что же делать? Воспользовавшись наследованием, можно определить класс с пустой структурой, а затем переопределить функции-члены так, как требуется для ваших файлов.

Также можно наделить этот класс способностью находить запись по заданному значению ее первого поля (действующего в качестве ключа). Например, можно будет задать файловому объекту поиск информации о сотруднике, порядковый номер которого равен 7.

В данном разделе представлена полная реализация класса textfile. Я надеюсь, что он будет полезен в ваших программах. Но главное, чтобы вы поняли, как обрабатывать файлы с помощью классов и как использовать старые наработки. Соответствующий этому классу код, равно как и функция fileopen(), включены в заголовочный файл c8tfile.h.

Для удобства в программу включен заголовочный файл nofranca.h, о котором рассказывалось на предыдущем уроке. Это дает два преимущества:

- Вы можете использовать функции askwords(), yesno() и т. д.
- Вы можете выбрать либо чисто текстовый интерфейс C++, либо графический интерфейс заголовочного файла franca.h. Если вам нужен текстовый интерфейс, то ничего делать не нужно. Если вам нужен графический интерфейс заголовочного файла franca.h, включите в свою программу директиву #include "franca.h" до того, как вы вставите туда файл c8tfile.h. Файлы franca.h и nofranca.h взаимоисключающие. Тот, который включен в программу первым, будет превалировать над другим.

Объявление класса

Ниже представлено объявление класса textfile:

```
class textfile
{
protected;
   fstream data;
   char datafile[40];
   int filemode;
```

```
pub! ic:
   char id[40];
   textfi le();
   textfile(char filename[], int mode=ios:: in|
                        ios::app|ios::nocreate);
   ~textfile();
   void displayC);
   vi rtual int read();
   int find(char id[]);
   int input();
   void write();
};
```

Данные-члены

Данные-члены защищены, чтобы они были доступны в производных от класса textfile классах:

- Объект data класса fstream нужен для выполнения операций над файлами.
- Переменная datafile представляет собой символьный массив, обозначающий имя дискового файла.
- В целой переменной filemode хранится информация о режиме открытия файла.

Имеется также открытая переменная id, обозначающая символьный массив, в котором хранится поле порядкового номера (ключ) рассматриваемой записи. Переменная открыта, поэтому каждый фрагмент программы может иметь доступ к записи.

Функции-члены

Основные функции-члены имеют следующее назначение:

- В классе textfile имеются два конструктора. Первым конструктором является конструктор по умолчанию без параметров. Он запрашивает имя используемого файла. У второго конструктора параметром является имя файла.
- *Деструктор* (destructor) ~textfile() — это специальная функция-член. В большинстве случаев особой необходимости в деструкторе класса не возникает. Деструкторы предназначены для удаления объекта и вызываются автоматически. В нашем случае с помощью деструктора выполняется закрытие файла. Действие деструктора противоположно действию конструктора.
- Функция display() предназначена для вывода на экран текущей записи.
- Функция input() предназначена для ввода записи с клавиатуры.

- Функция find() предназначена для поиска записи в файле по первому полю (ключу), содержание которого и является параметром функции. **Отметьте,** что любой тип данных в текстовом файле представлен символьной строкой. Единственное вводимое нами ограничение касается размера (39 символов) и положения ключа (он должен находиться в первом поле записи).
- Функция write() предназначена для сохранения в файле текущей записи.
- Функция read() предназначена для считывания записи с диска.

Конструкторы

Ниже представлен программный код конструкторов:

```
#include "nofranca.h"
textfile::textfile()
{
   askwords(datafile, 40, "enter the filename:");
   filemode=ios:: in | ios::app | ios::nocreate;
   data.open(datafile, filemode);
   if(fileopen(data)==0)
   {
      if(yesno("File not open, create?"))
      {
         filemode=ios::out;
         data.open(datafile, filemode);
         if(fileopen(data)==0) // Проверка факта открытия файла
         {
            yesno("Sorry, file does not open!");
            exit(0);
         }
         else
      }
      else exit(0);
   }
}
textfile::textfile(char filename[], int mode)
{
   if(sizeof(datafile)>>strlen(filename))
      strcpy(datafile, filename);
```

```
    data.open(datafile, mode);
    filemode=mode;
    if(fileopen(data)==0)
    {
       yesno("Error in opening, will you check?");
       exit(0);
    }
}
```



СОВЕТ Функции, представленные в предыдущем фрагменте кода, рассчитаны на заголовочный файл nofranca.h, который позволяет работать с функциями askwords() и yesno(). Для перехода в графический интерфейс необходимо перед тем, как вставить файл c8tfile.h, воспользоваться директивой #include "franca.h". Не забудьте также, что при использовании заголовочного файла franca.h основной функцией программы будет не main(), а mainprog().



ПРИМЕЧАНИЕ С заголовочным файлом franca.h нельзя использовать объекты cin и cout, а нужно вернуться к информационным рамкам.

Функция exit(O) предназначена для обработки экстремальной ошибочной ситуации, при возникновении которой она завершает работу программы.

Конструкторы находят файл по заданному имени и открывают его для ввода и добавления информации.



ПРИМЕЧАНИЕ Выражение ios::in|ios::app может показаться вам необычным. Вертикальная черта представляет собой логический оператор дизъюнкции. Выражение в целом позволяет открыть файл как для ввода, так и для добавления информации.

Деструктор

Код деструктора также достаточно прост:

```
textfile::~textfile()
{
   data.close();
}
```

Ввод с клавиатуры и вывод на экран

Коды функций display() и input() очень интересны:

```
void textfile::display()
{

}
inttextfile::input()
{
   askwords(id, 40, "Enter id code");
   if(id[0]==0) return 0;
   return 1;
}
```

Как работают эти функции? Функция display(), например, вообще ничего не выводит! Правильно ли это? А где идентификационные номера и почасовая оплата?

Помните, нельзя решить все проблемы сразу. Если включить в ваш класс имена, идентификационные номера и другую информацию, то его нельзя будет использовать повторно! (Например, в других приложениях, касающихся продажи автомобилей, учета рабочих мест и т. д.) Таким образом, класс textfile должен стать базовым для будущих производных классов, а функция display() в каждом из этих производных классов будет заменяться новой версией. Так вы сможете выводить на экран все данные-члены в требуемом формате.

Функция input() запрашивает ввод данных для поля идентификационного номера. Впоследствии вы можете построить на ее основе новую версию функции, которая после ввода идентификационного номера будет продолжать вводить остальные данные-члены. Можете также полностью перегрузить исходную версию функции input().

Представленная выше функция input() запрашивает ввод данных с клавиатуры и контролирует ввод нулевой строки (например, просто нажатие клавиши Enter). Это один из способов показать, что вы больше не собираетесь вводить данные. Затем функция возвращает 1 или 0. Если моя идея вам подходит, пользуйтесь. Если нет, придумайте в производном классе что-нибудь свое.

Считывание и запись

Рассмотрим код функций-членов read() и write():

```
int textfile::read()
{
   // Считывание очередной записи из файла:
   data» id;
```

```
    if(data.eof()) return 0;
    return 1;
}
void textfile::write()
{
    data.close();
    data.open(datafile, ios::app);
    if(fileopen(data)==0)
    {
       yesno("Failed reopen for append, will you check?");
       retutn;
    }
    data<<endl<<id<< ;
    data.closeO;
    data.open(datafile, filemode);
}
```

Функция read () просто считывает из файла очередной идентификационный номер. Ее тоже нужно дополнить, чтобы другие поля записи также могли считываться из файла.

Функция write() действует аналогично, за исключением того, что она сначала закрывает файл, а затем вновь открывает его для вывода. Обратите внимание, что после идентификационного номера для разделения полей в текстовом файле ставится пробел. После записи идентификационного номера файл закрывается и затем снова открывается в том же режиме. В производном классе эту функцию-член также нужно дополнить.

Поиск записи в файле

Ниже представлен программный код функции-члена find():

```
int textfile: :find(char idnumber[])
{
    data.closeO;
    data.open(datafile, filemode);
    if(fileopen(data)==0)
    {
       yesno("Failed reopen forfind, wiI you check?");
       exit(0);
    }
    for(;;)
```

```
    {
       if (data.eof())break;
       if(stromp(id, idnumber)!=0) read(); // Виртуальная функция!
       else return 1;
    }
    data.close();
    data.open(datafile, filemode);
    return 0;
}
```

Эта функция просто считывает все записи, сравнивая их ключевое поле с требуемым идентификационным номером. Если такая запись найдена, функция возвращает 1. В противном случае она возвращает ноль. Поиск каждый раз ведется от начала файла. Возможно, это и не самый лучший вариант, но зато достаточно простой. Он требует закрытия и затем повторного открытия файла, чтобы поиск всегда велся с начала файла.

Важнейшим аспектом функции find() является считывание записей. Каждая запись считывается из файла посредством вызова функции-члена read().

Обратите внимание, что имеется стандартная функция read() исходного класса textfile и другая функция read(), которую вы должны написать для производного класса. О какой же функции read() идет речь сейчас?

Если вы пользуетесь стандартной функцией read(), определенной для класса textfile, то будет считано только поле идентификационного номера, поэтому следующие попытки приведут к использованию в качестве ключа имени или почасовой оплаты. Причина в том, что считывание из файла начинается с того поля, на котором закончилось предыдущее считывание.

Например, пусть ваш производный класс считывает записи, состоящие из идентификационного номера и имени, с помощью функции-члена read() из базового класса. Но в этой исходной функции readQ нет информации о поле имени, поэтому при считывании записи будет считано только поле идентификационного номера. Когда считывание будет продолжено, компьютер начнет с того места, где он остановился, и считает часть имени, приняв его за следующий идентификационный номер.

Использование вместо исходной функции read() собственной версии этой функции, определенной для соответствующего производного класса, является обязательным правилом. Именно по этой причине функция read() в классе textfiLe объявлена виртуальной.

Класс textfiLe вы найдете в заголовочном файле c8tfile.h прилагаемой дискеты.



ПРИМЕЧАНИЕ Передавать функции объекта класса textfile а качестве аргумента можно только по ссылке. При передаче аргумента по значению создается его копия, которая затем удаляется. Но удаление копии ведет к закрытию файла.

Самостоятельная практика

О Для работы с файлом c8persn1.txt создайте класс workers, производный от класса textfile. Напишите программу, позволяющую вывести на экран содержимое файла в виде списка.

Базовые операции с файлами

Вы должно быть уже заметили, что файлы являются великолепными объектами для хранения информации, предназначенной для многократного использования. Благодаря им отпадает необходимость несколько раз вводить в компьютер одни и те же данные. Более того, большие объемы данных иногда просто невозможно набрать заново. На примере предыдущего раздела понятно, насколько удобно хранить информацию о сотрудниках в файле.

Отметьте, что идентификационный номер сотрудника и его имя предполагаются неизменными. Почасовая оплата, конечно, может меняться, но вряд ли это будет происходить слишком часто. По этой причине информация, хранящаяся в файле, является в некотором роде постоянной.

Класс textfile обеспечивает базовые функции для работы с файлами:

О Ввод данных с клавиатуры.

О Запись данных в файл.

О Считывание данных из файла.

О Вывод данных на экран.

О Поиск и считывание из файла требуемой записи.

Пользуясь этим классом, можно включать в файл новых сотрудников, контролировать данные об уже существующих и т. д. Однако в реальной ситуации может потребоваться поддержка и других операций с файлами.

О Что делать, если сотрудник покидает компанию? Необходимо научиться удалять информацию из файла. То есть необходима операция *удаления записи*.

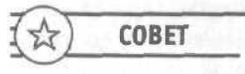
О Что делать, если сотруднику нужно повысить оплату? Необходимо научиться находить соответствующую сотруднику запись, изменять значение оплаты и возвращать запись на прежнее место. То есть необходима операция *обновления записи*.

Обновление данных в файле

К сожалению, для текстовых файлов выполнение операции *обновления* данных может оказаться весьма не простым, поскольку размеры записей о разных сотрудниках разные. На экране каждая запись кажется отдельной строкой. Но на самом

деле в файле все записи расположены непрерывно, одна за другой. Поэтому если размер обновленной записи превысит свое прежнее значение, ее нельзя будет поместить на прежнее место. Кроме того, компьютер начинает каждую следующую операцию считывания данных из файла с той позиции, на которой закончилась предыдущая. Необходимо гарантировать, что после считывания обновленная запись займет свое прежнее место.

Тем не менее в текстовых файлах можно реализовать операции обновления данных. Самое простое решение — это создании второго файла и сохранение в нем обновленной информации. То есть нужно записывать считанные данные не в старый, а в новый файл. С другой стороны, если сделать размеры всех записей одинаковыми, с бинарными файлами было бы работать несколько легче.



СОВЕТ В любом случае вы выиграете, если сделаете одинаковым размер всех записей в своем файле.

Соответствие данных в файлах

Другой важной операцией, которая может потребоваться для работы с файлами, является поиск *соответствия* (matching). Эта операция используется при работе с двумя или более файлами, когда по записи из одного файла ищется соответствующая ей запись в другом файле.

Рассмотрим, например, файл workhours, в котором содержится идентификационный номер сотрудника и число отработанных им часов. Для вычисления причитающейся ему зарплаты необходимо по номеру найти запись из файла workhours и соответствующую ей запись из файла worker, в котором содержится имя и почасовой оклад данного сотрудника. Рассчитать заработную плату сотрудника можно, только имея в руках ту и другую информацию.

Благодаря классу textfile операция поиска соответствующих записей в файлах упрощается, поскольку функция-член find() может находить документ по заданному идентификационному номеру.

В следующем примере мы будем создавать платежную ведомость. Пусть имеются два файла:

- Файл c8persnl.txt (файловый объект payroll) содержит идентификационный номер сотрудника, его полное имя и почасовую оплату.
- Файл c8payrll.txt (файловый объект worker) содержит идентификационный номер сотрудника и число отработанных им часов.

Наша задача — вывести полный список сотрудников, их персональные данные и причитающуюся каждому сумму. Поскольку почасовая оплата и отработанное время находятся в записях из разных файлов, то и соответствующие записи о каждом сотруднике придется искать в разных файлах.

Можно брать запись из файла payroll и искать соответствующую ей запись в файле worker, а можно, наоборот, брать запись из файла worker и искать соответствующую ей запись в файле payroll. Однако лучше считывать информацию из того файла, в котором она чаще меняется (в данном случае это файл payroll) и периодически вносить изменения в файл, информация в котором относительно постоянна (файл worker).

С каждой записью, взятой из файла payroll, необходимо выполнить следующие операции:

- Получить идентификационный номер сотрудника и количество отработанных им часов.
- В файле worker найти запись с тем же идентификационным номером.
- Вычислить зарплату данного сотрудника.

И конечно же, нужно решить, что делать, если соответствия не найдено. Мы существенно облегчим поиск соответствия, если для работы с файлом worker воспользуемся классом hired_person, поскольку в этом случае поиск записи о сотруднике по его идентификационному номеру можно выполнить с помощью функции-члена flnd().

Файл payroll нужно определить в классе, производном от класса textfile. Это просто, поскольку в новый класс нужно включить *только* одну переменную — количество отработанных часов — и одну функцию-член для считывания записей из файла. Код для функций вывода на экран, записи, ввода и поиска уже готов!

Класс payfile

Одна из возможных реализаций класса для обработки файла payroll имеет следующий вид;

```
#include "nofranca.h"
class payfile:public textfile
{
public:
   float hours;
   virtual int read();
};
int payfile::read()
{
   if (textfile::read()==0) return 0;
   data>>hours;
   return 1;
}
```

Функция read() класса payfile вызывает базовую функцию read() класса textfile. Чтобы показать, что используемая вами функция-член не относится к текущему классу, нужно полностью описать (то есть уточнить) ее имя. Как мы знаем, это делается с помощью последовательного задания имени класса, двух двоеточий и имени функции:

```
имя_класса::имя_функции();
```

В нашем примере аналогичная инструкция выглядит следующим образом;

```
if (textfile::read()==0) return 0;
```

Когда доступны файлы обоих классов, основная функция программы создания платежной ведомости не выглядит слишком сложной:

```
void main()          //c8payr11.cpp
{
   // Объявление переменных:
   char filename[40];
   cout<<"Opening the employees file:";
   hired_person worker;
   cout<<"Opening the payroll file:";
   payfileworkhours;
   float paycheck, payroll=0;
   char  idnumber[40];
   // Цикл для первого файла:
   for(;;)
   {
      // Считывание записи из файла workhours,
      // если не достигнут конец файла:
      if(workhours.read()==0) break;
      // Поиск данных о сотруднике а файле worker:
      if(worker.find(workhours.id))
      {
         paycheck=worker.wage*workhours.hours;
         payroll=payroll+paycheck;
         worker.display();
         cout<<" $"<<setw(12)
            «set iosflags(ios::fixed)«paycheck;
      }
      else
         cout<<endl<<"No record found for worker:"
            <<worker.id;
   }
   cout<<endl<<"Payroll total: $"<<setw(12)<<payroll;
}
```

Самостоятельная практика

О Создайте для файла payroll функции-члены input(), write() и display(). В вашей реализации все записи должны быть одинакового размера (при необходимости для заполнения пустого пространства используйте пробелы).

Что нового мы узнали?

В этом уроке мы научились

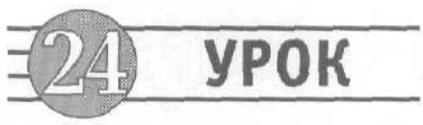
0 Хранить данные в файлах.

0 Использовать директивы препроцессора tfdefine, tfifdef, ttifndefи tfendif.

0 Создавать специальные классы для обработки файлов.

0 Считывать данные из файла и записывать данные в файл.

0 Создавать приложения для обработки нескольких файлов.



Создание реального торгового терминала

- Совершенствование торгового терминала
- Хранение каталога продуктов в файле на диске
- Многократное использование классов

В русле дальнейшего развития наших навыков создания приложений мы усовершенствуем проект торгового терминала, чтобы максимально приблизить его к реальности. Код продукта в наш новый терминал будет вводиться с клавиатуры, а цена и описание продукта будут сохраняться в файле.

Возвращение к проекту торгового терминала

Созданный на уроке 21 торговый терминал был достаточно несовершенен, поскольку при каждом запуске программы приходилось вводить весь каталог продуктов заново. Избежать этого можно, если хранить данные не в оперативной, а в дисковой памяти.

В данной версии используется тот же самый терминал, но теперь каталог продуктов для него создается в файле на диске.

Реализация новых возможностей

Если внимательно изучить последнюю версию торгового терминала, можно заметить, что класс catalog инкапсулирует все необходимые для каталога операции. Фактически, если переделать класс catalog таким образом, чтобы данные хранились не в массиве, а в файле, проблема будет решена.

Единственное, что для этого нужно, это создать еще один альтернативный класс catalog!

Поскольку вы уже без сомнения стали энтузиастом наследования, возможно, захотите создать из старого класса catalog новый производный класс и затем использовать его в своей версии терминала. Можно поступить и так.

Однако исходный класс catalog содержит несколько совершенно лишних атрибутов. Например, зачем нам массив, если все данные все равно хранятся в файле? Поэтому в нашем случае класс catalog лучше не наследовать, а переделать.

Абстрактные классы

Чтобы в дальнейшем использовать оба класса catalog, можно проделать следующий трюк: объявить класс catalog, все функции которого являются виртуальными (как функция-член display класса textfile), а затем создать из него два производных класса, например arraycatalog и filecatalog. Оба этих класса смогут обрабатывать каталоги продуктов, но каждый из них будет это делать по-своему. В этом случае создание объекта базового класса невозможно в принципе, это так называемый *абстрактный класс* (abstract class).

Создание класса catalog

Класс catalog легко сделать производным от класса textfile. Рассмотрим следующее объявление:

```
struct product
{
   char code[40];
   char description[40];
   float price;
};

class catalog:public textfile
{
   product saleitem;
   virtual int read();
   word write();
public:
   virtual product find(char part_number[]);
   virtual product find(int somecode);
};
```

Две функции find()

Возможно, вас удивило, что мы объявили две функции с одним именем find(). На самом деле классу textfile для обработки символьного массива (ключа) нужна всего одна функция. Однако в нашем пока единственном работающем торговом терминале в качестве ключа используется целое значение. Эта новая функция позволяет согласовать работу обоих терминалов без изменения кода!

Виртуальные функции

Хотя мы начали знакомство с виртуальными функциями на уроке 17 и затем продолжили его в классе textfile, сейчас следует снова к ним вернуться.

Наши первые виртуальные функции позволяли двигаться бегуну и конькобежцу, Но каждый из них делал это по-своему. Рассмотрим, как выполнялись наши прежние инструкции:

```
runner julia;
skater mike;
julia.run();
mike.run()
```

Этот фрагмент программного кода благополучно компилировался. При этом компилятор гарантировал, что с объектом julia будет использована функция run(), определенная для бегуна, а с объектом mike — функция run(), определенная для конькобежца. Следовательно, когда у компилятора имеется информация, для объектов какого типа определена функция, он правильно вызывает нужную версию функции.

Однако в некоторых ситуациях компилятор не может определить класс используемого объекта, поскольку его истинная природа проявляется уже в процессе выполнения программы, когда компиляция закончена!

На уроке 17 рассматривалась функция, параметром которой был объект класса runner:

```
void march(runner volunteer)
{
   volunteer.run();
}
```

Вся проблема в том, что аргументом функции может быть как бегун, так и конькобежец (который благодаря наследованию тоже является бегуном). Следовательно, функция может быть вызвана как с тем, так и с другим. Но компилятор может связать объект только с одной версией функции run(). Однако у него нет информации, кого вы хотите выбрать, бегуна или конькобежца (или объект любого другого, созданного вами класса).

С другой стороны, когда вы определяете виртуальную функцию, то компилятор откладывает реальное связывание кода с нужной функцией до этапа выполнения программы. Это называется *поздним связыванием* (late binding).

Чтобы реализовать абсолютный полиморфизм, в идеале все функции должны быть виртуальными. Почему тогда мы их так не определяем?

Из практических соображений. Необходимый для позднего связывания код делает ваши программы длиннее и медленнее, поэтому, если компьютер знает, как определить класс объекта, виртуальные атрибуты на самом деле не нужны. Таким образом, C++ заставляет программиста самому решать, для каких функций необходимо позднее связывание.

Интересная ситуация, требующая позднего связывания (виртуальных функций), возникает в классе textfile. В некоторый момент для считывания следующей записи функция find() вызывает функцию read(). Не забывайте, что исходная функция

read{) считывает только поле с идентификационным номером, а для считывания остальных полей были задуманы другие функции. Но нам требуется считывать всю запись, а не только идентификационный номер, а при этом внутри функции find() определить тип связываемого объекта на этапе компиляции невозможно. Ничего страшного, поскольку функция read() виртуальная, тип объекта выяснится уже в процессе выполнения программы.

Вывод на экран и ввод с клавиатуры

В производных от класса textfile классах должны быть собственные версии функций ввода и вывода. Где же они?

Нигде! На самом деле вам не нужно вносить данные в каталог и выводить их на экран. Ввод и проверку данных можно выполнить в любом текстовом процессоре. В файл c8parts.txt уже включено несколько пунктов каталога. Вы можете воспользоваться ими или добавить в каталог несколько своих пунктов.

Конструктор

Задавать конструктор не нужно. Можно воспользоваться стандартным конструктором класса textfile. Он прекрасно выполнит задачу. Сначала запросит имя требуемого файла, а затем и откроет его.

Функции read() и write()

Функции read() и write() передают содержимое текущей записи структуры saleitem в файл и обратно. Ниже представлены коды этих функций:

```
int catalog::read()
{
   int testeof=1;
   testeof=textfile::read();
   if(testeof==0) strcpy(id, " ");
   strcpy(saleitem.code, id);
   data>>saleitem.description;
   data>>saleitem.price;
   return testeof;
}
void catalog::write()
{
   textfile::write();
   data<< «saleitem, description;
   data<< <<saleitem.price;
}
```

Обратите внимание, что в обеих функциях имеются вызовы их версий для базового класса (textfile::read() и textfile::write()). Это совсем не обязательно, но тем не менее использовать готовый программный код всегда полезно. В этом случае код продукта будет продублирован, поскольку класс catalog наследует буквенный массив id от класса textfile, и код продукта определяется в структуре. Поэтому вне зависимости от того, считана информация в структуру или нет, код продукта все равно появится в массиве *id*.

Поиск записей

Ниже приводятся функции поиска записи по заданному идентификационному коду продукта:

```
product catalog: :find(char part_number[])
{
   int kfind;
   kfind=textfile::find(part_number);
   if  (kfind==0)
   {
      saleitem.code[0]=0;
      id[0]=0;
   }
   return saleitem;
}
product catalog::find(int somecode)
{
   char thecode[40];
   itoa(somecode, thecode, 10);
   return find(thecode);
}
```

Код всего проекта можно найти в файлах c8catlg.h и c8term.h прилагаемой дискеты. В основной программе c8term.cpp используется заголовочный файл franca.h и графический интерфейс Windows.

```
// Программа              //c8term.cpp
// Для выбора компилятора (Borland или Microsoft) удалите
// символы комментирования с одной из двух следующих директив:
//    ttdefine    Microsoft
//     #define Borland
ffinclude "franca.h"
#include "c8catlg.h"
```

```
ttinclude "c8term.h"
void mainprog()
{
   saleterm cashregister;
   cashregister.operate();
}
```

Для проверки правильности открытия файлов необходимо выбрать подходящую для своего компилятора директиву tfdefine. Эту программу можно также использовать со стандартным (не графическим) интерфейсом. Для этого нужно удалить инструкцию tfindude "franca.h" и изменить основную функцию mainprog() на main().

Что нового мы узнали?

В этом уроке мы научились

0 Обрабатывать каталог продуктов в файле на диске.

0 Повторно использовать определенные ранее классы.

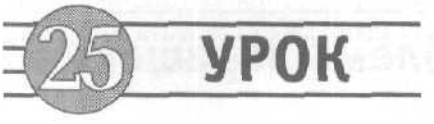
Часть IX

Проблемы и решения

В части IX вы познакомитесь с наиболее распространенными проблемами, с которыми обычно сталкиваются программисты, и на основе полученных знаний об объектно-ориентированном программировании разработаете еще одно приложение.

Необходимо четко понимать, что с помощью одной, даже очень хорошей книги невозможно узнать *все* о программировании вообще и о C++ в частности. Поэтому мы обсудим те направления, в которых вам надо работать, чтобы и дальше развивать свои программистские навыки.

Если вы уже почувствовали себя знающим программистом и нуждаетесь в дополнительной информации, воспользуйтесь книгой Дж. Элджера «C++: библиотека программиста» (издательство «Питер»). Если вас интересует создание приложений в новой визуальной среде разработки Borland C++ Builder, вам поможет книга М. Теллеса «Borland C++ Builder: библиотека программиста» (издательство «Питер»).



Создание объектно-ориентированного приложения

- Объектно-ориентированное программирование
- Многократное использование программного обеспечения
- Разработка небольшого проекта электронной игрушки
- Реализация объектно-ориентированной версии проекта

На этом уроке будут обобщены основные приемы объектно-ориентированного программирования (Object Oriented Programming, OOP), и на базе этих приемов намечен, проработан и реализован короткий проект.

Рекомендации, которые вы здесь встретите, не нужно понимать буквально. Имеются различные точки зрения на то, как нужно создавать и отлаживать программное обеспечение. Вы лишь должны принять во внимание изложенный здесь материал и в дальнейшей работе руководствоваться здравым смыслом.

Объектно-ориентированный менталитет

Объектно-ориентированное программирование дает программисту мощный инструмент в его профессиональной деятельности. Однако любой новый инструмент бесполезен, пока вы не научитесь с ним работать. Никогда не забывайте, что созданный сегодня фрагмент программного кода всегда может пригодиться завтра. И чем больше будет таких фрагментов, тем больше времени и сил вы сэкономите! Старайтесь максимально упростить использование, расширение возможностей (лучше на базе наследования) и обслуживание своего программного обеспечения.

Хороший программист может разрабатывать работоспособные и надежные программные компоненты в предельно короткие сроки. Как ему это удается? Конечно, здесь помогают знания и опыт, но что еще?

Благодаря многократному использованию программного обеспечения можно сэкономить массу времени на его создание и обслуживание. В следующих разделах рассматриваются два интересных аспекта этого вопроса;

- Многократное использование доступного программного обеспечения.
- Разработка программного обеспечения с прицелом на его многократное использование.

Имеются и другие, более ограниченные пути использования готовых фрагментов программного кода — это их переделка и доработка.

Переделка, доработка и многократное использование программ

Допустим, вы написали фрагмент программы для считывания слов с клавиатуры и сохранения их в файле. Через некоторое время вы получаете заказ на разработку программного обеспечения, в котором, кроме всего прочего, требуется решить и эту задачу.

Программист может зыбрать один из следующих вариантов действий:

- Сесть перед компьютером и написать нужный фрагмент. В конце концов это займет всего несколько часов! Вы, вероятно, осознаете, что это далеко не лучший путь. Однако так поступает большинство программистов. Поскольку для такого варианта действий нет готового термина, назовем его *переделкой программы*.
- Вспомнить, что аналогичный фрагмент программы уже разработан. После этого остается найти его на диске, скопировать в новую программу и ввести необходимые изменения. Назовем это *доработкой программы*.
- Если предыдущий фрагмент специально разрабатывался с учетом возможности его многократного использования, то он, вероятно, уже хранится в заголовочном файле. Поэтому программист может просто включить этот файл в свою новую программу. Назовем это *многократным использованием программы*.
- Имеется и еще один вариант. Хотя предыдущий фрагмент и годен к многократному использованию, он не совсем подходит для решения новой задачи. Тем не менее если в своей новой программе программисту удается включить и использовать этот фрагмент без изменений (например, путем создания в новой программе производных классов), то по-прежнему можно говорить о *многократном использовании программы*.

Переделка или доработка

К сожалению, работа большинства программистов сводится к переделке программ. В конце концов, как они считают, их дело писать новые программы, а не копаться в старых. Переделка несет в себе двоякий вред:

1. Память программиста обычно не столь надежна, как ему кажется. На переделку программы может уйти столько же времени, сколько когда-то ушло на создание аналогичной версии. И даже при повторном написании программы могут появиться жучки (часто те же, что и раньше), тоже ведущие к потере времени.
2. Если продолжать действовать в том же духе, то в конце концов появятся проекты, в которых одна и та же задача решается в разных и далеко не всегда полностью идентичных фрагментах кода. В процессе обслуживания программы все такие фрагменты приходится находить, разбираться в них и при необходимости изменять.

Доработка в значительной степени сводит на нет первый недостаток переделки, поскольку программист сразу переходит к проверенной версии программного обеспечения. Но из-за того, что эта проверенная версия подвергается изменениям, второй недостаток переделки по-прежнему остается в силе.

Многократное использование программ

Многократное использование фрагментов программного кода существенно облегчает обслуживание готового программного обеспечения. Модернизировав предназначенный для многократного использования фрагмент программного кода, его

относительно легко включить в те программные проекты, в которых аналогичные функции выполнял прежний фрагмент. Таким образом, любое усовершенствование такого фрагмента непосредственно сказывается на всем программном обеспечении.

Что мы понимаем под многократным использованием программ? Это ситуация, в которой один и тот же фрагмент кода неоднократно включается в разные программные продукты. Это может произойти в одной программе, в одном пакете программ или даже в совершенно не связанных между собой программах.

На самом деле многократное использование программного обеспечения возникло с возникновением функций (или, как их раньше называли, подпрограмм). Чтобы не разрабатывать определенные фрагменты кода снова и снова, программисты для ускорения своей работы научились работать с функциями, которые сохраняли в специальных библиотеках. Многократное использование программ не только ускоряет программирование, но и повышает его надежность, поскольку чем чаще и в более разных условиях выполняется тот или иной фрагмент программы, тем более полной проверке он подвергается.

Объекты поднимают многократное использование программ на новую высоту. Если вы, даже имея целую библиотеку функций, нуждаетесь в несколько иной версии, то вся библиотека может оказаться бесполезной. Вам придется либо модернизировать близкую по назначению функцию (вводя в нее непроверенный код), либо разрабатывать совершенно новую. Кроме того, любая функция предназначена для обработки данных. Но при этом никакого способа защитить эти данные от случайного изменения функция не предлагает.

Классы позволяют ограничить число допустимых типов данных и благодаря наследованию модифицировать часть предназначенных для их обработки операций.

Широкое применение механизма многократного использования программ требует строгой дисциплины. Много времени может отнимать тестирование соответствующих фрагментов. Более того, в некоторых библиотеках классов настолько сложно разобраться, что программист может впасть в искушение найти собственное решение, а не тратить время, пытаясь понять, как ту же задачу решил кто-то другой. Здесь нужно руководствоваться здравым смыслом. На разработку собственного фрагмента иногда действительно требуется меньше времени, чем на изучение подходящей программы.

Проблема в обслуживании. Чем больше в программе готовых фрагментов кода, тем меньше времени требует ее обслуживание. Это все равно, что содержать в порядке механизм, собранный из стандартных деталей!

Интересный компромисс состоит и в том, чтобы создавать свою программу (если вы все таки решились на это) также в расчете на многократное использование.

Инвестиции вместо расходов

Разработка собственной библиотеки классов поначалу будет отнимать много времени, поскольку кроме создания задела на будущее вам придется решать текущие

задачи. Но это будут ваши инвестиции в будущее. По мере роста числа поставленных задач вы постепенно начнете пользоваться своей библиотекой и, соответственно, быстрее решать эти задачи!

При этом не забывайте, что с каждым разом ваши программы будут становиться проверенный и надежней.

Некоторые программисты считают, что они очень много «многократно используют» только потому, что работают с готовыми библиотеками классов. Но этого не достаточно! Они по-прежнему тратят время на то, что они уже когда-то делали. Полезно, когда программист ленится стучать по клавиатуре, но лениться думать ему непозволительно.

Получив задание, всегда загляните вперед. Возможно, с аналогичной проблемой вы встретитесь снова. Может быть, для решения проблемы полезно создать класс? Может быть, сделать этот класс легко настраиваемым? Может быть, его стоит включить в свою библиотеку классов?

Проект электронной игрушки

В этом разделе мы рассмотрим проект и его реализацию. Для этого ваш компьютер должен иметь звуковую карту. Цель проекта — создание электронной игрушки, развивающей у детей навыки правописания.

Из компьютера вы услышите слово, которое предварительно было введено и записано в файле. Далее в хорошо вам знакомой информационной рамке игроку требуется набрать услышанное слово. Если его слово написано правильно, игроку об этом сообщается, и он получает очко. При этом также учитывается время, затраченное на обдумывание.

Эта процедура повторяется с несколькими словами. Когда список слов заканчивается, появляется табло, на котором игрок может увидеть количество правильно набранных слов, количество ошибок и затраченное на обдумывание время.

Помимо этого, программа может заранее сообщить все слова и показать их правописание.

Предполагаемая реализация

Для записи слов в отдельный файл используется звуковая карта. Для простоты ограничимся словами, содержащими не более восьми букв.

Список слов можно хранить в символьном массиве в оперативной памяти или в файле на диске. Рассмотрите возможность хранения их на диске и последующей загрузки в массив.

Набрать и вставить список слов в файл можно с помощью любого текстового процессора. Позднее ввод слов с клавиатуры можно предусмотреть непосредственно в программе.

Процедура

Ниже представлено описание алгоритма:

1. Задать переменную wordnumber — число доступных слов (и звуковых файлов).
2. Задать переменную testnumber — число проверяемых слов (обычно от 6 до 20).
3. Загрузить слова из файла в массив.
4. Для каждого слова в массиве:
   - На несколько секунд показать слово в информационной рамке.
   - Произнести его вслух.
5. Повторить следующие пункты столько раз, сколько слов задано в переменной testnumber:
   - Выбрать произвольное число от 1 до значения переменной wordnumber.
   - С помощью звуковой карты сообщить слово, находящееся в массиве на позиции wordnumber–1.
   - Включить таймер.
   - Вывести на экран запрос на ввод слова.
   - Засечь время.
   - Проверить правильность написания слова и сообщить об этом игроку.
   - Обновить значение счетчика очков.
6. Сообщить игроку счет и затраченное время.

Усовершенствования

Когда программа будет составлена и успешно пройдет апробацию, ее можно будет усовершенствовать:

О Исключить повторение правильно написанных слов.

О Иллюстрировать процесс набора очков картинками.

Можно, например, при каждом неправильном ответе рисовать по частям какого-нибудь страшного монстра или складывать монеты в две стопочки: в одну при правильных ответах, а в другую при неправильных.

Пользовательский интерфейс

В данной ситуации пользовательский интерфейс может оказаться ключевым фактором, определяющим успех вашего программного продукта. Если вы работаете в расчете на малышей, то заинтересовать ваших пользователей могут только хорошие иллюстрации. Если же вы забудете об этом, ваша игра надоест им в два счета!

На рис. 25.1 показан экран компьютера после одного сеанса игры. Каждая зеленая монетка (в левой стопочке) соответствует правильному ответу, а каждая красная

(в правой стопке) — неправильному. Кроме того, когда пользователь правильно набирает слово, раздается сообщение: «All right!»

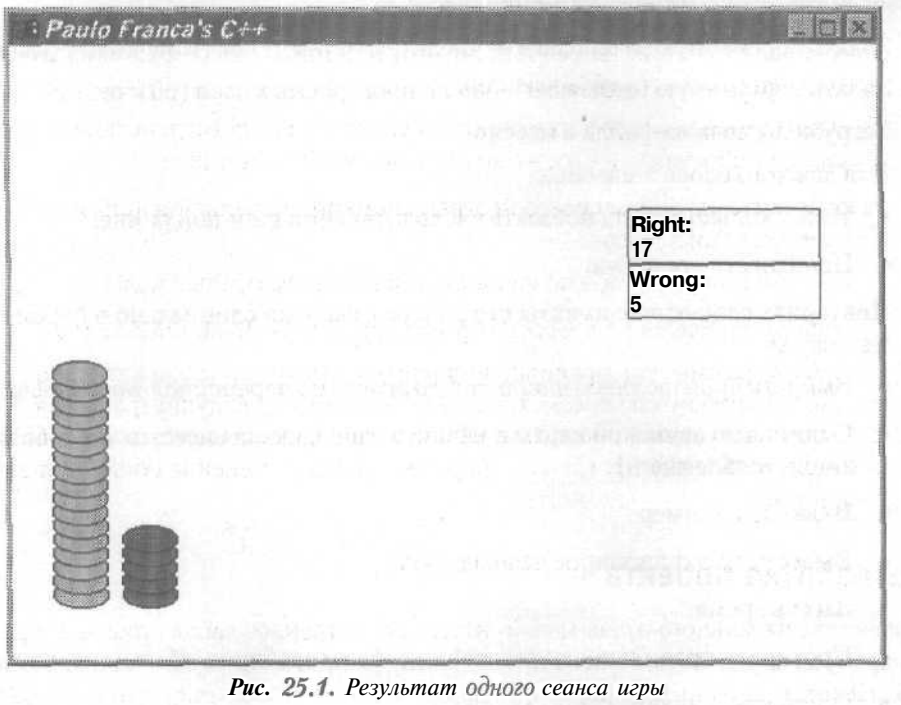


Рис. 25.1. *Результат одного сеанса игры*

Объектно-ориентированная реализация

Для разработки проекта программного обеспечения на базе объектно-ориентированного подхода не нужно придерживаться никаких специальных правил. Тем не менее некоторые советы будут вам полезны:

1. Никогда не бойтесь начать. Сформулируйте, как *можно* было бы решить проблему. Еще и еще раз проанализируйте решение. Измените и дополните, словом постарайтесь хорошо описать то, как вы собираетесь решать проблему.
2. Делите программу на фрагменты, пока объем каждого из них не станет максимально удобным для работы. Если вы собираетесь реализовать механизм многократного использования фрагментов программного кода, то неважно, будут это готовые фрагменты или вы создадите свои.
   - Помните, что при многократном использовании любого фрагмента вы сберегаете массу времени и, скорее всего, получаете более надежное решение. Необходимо подобрать доступный вам класс объектов, который мог бы оказаться полезным, а если такой класс недоступен, разработайте собственный, но с учетом возможности его многократного использования в будущем.

- Создавая новые классы, делайте свое решение по возможности более общим и дополняемым. Не решайте только сегодняшние проблемы. Заранее готовьтесь к тому, что может случиться завтра.

3. Сразу после того, как вы решили, что тот или иной класс может оказаться полезным, напишите его объявление.
   - Объявления классов устанавливают способы их взаимодействия с другими частями программы. Сам же код можно написать позднее.
4. Разработайте алгоритм своего программного обеспечения, используя объекты выбранных вами классов.
   - На этом этапе уже можно начинать писать программный код.
5. Напишите определения классов — реальный код его функций-членов.

Создание программы, как правило, не сводится к такому последовательному процессу. Часто приходится пересматривать некоторые предыдущие решения. Имеет смысл делать все возможное, чтобы ваш код стал более надежным и удобочитаемым. После появления программы на рынке любое изменение обойдется вам гораздо дороже!

Разработка проекта

В качестве начального этапа можно взять рассмотренное выше описание проекта. Попробуем определить, какие объекты могли бы пригодиться для решения проблемы. Некоторые из них, такие как информационные рамки или экранные объекты, имеются в заголовочном файле franca.h и, несомненно, будут нам полезны. Но этих объектов очевидно недостаточно. Нужно ли нам искать новые классы объектов?

Если мы попытаемся разложить проблему на многократно используемые фрагменты (в соответствии с пунктом 2 предыдущего раздела), то в результате можем прийти к набору объектов, с помощью которых цель будет достигнута.

Если перенести нашу игровую ситуацию на реальную жизнь, то можно вообразить себе учителя, который диктует своим ученикам список слов, взятых по всей видимости из словаря.



ПРИМЕЧАНИЕ Итоговый код этого проекта можно найти в файле c9spell.cpp прилагаемой дискеты. Звуковые файлы находятся там же в подкаталоге sounds.

Обработка слов

Описанная только что воображаемая ситуация побуждает нас использовать объект класса teacher (учитель), который будет диктовать слова и заниматься проверкой. Такой объект должен уметь выполнять следующие основные операции:

- Выводить все проверяемые слова на экран.
- Озвучивать слова.
- Проверять слова.

Список слов можно взять откуда угодно, например из словаря. Было бы интересно, если бы учитель мог выбирать слова из разных списков. Кроме того, ввиду подключения звука, прекрасно было бы придумать объект, совмещающий в себе словарь и диктофон. В таком объекте классаjukebox (звуковая библиотека) должен храниться список слов (как в обычном, так и в звуковом формате), и он должен уметь:

- Правильно озвучивать заданное слово.
- Правильно писать заданное слово.
- Сообщать количество имеющихся в списке слов.

Учитель, несомненно, будет диктовать и проверять свои слова с помощью такой звуковой библиотеки.

Реализация классаjukebox, вероятно, окажется весьма полезной, поскольку вообразить объект, озвучивающий слова из некоторого списка, можно в самых разных ситуациях.

Поскольку нам часто придется иметь дело с символьными массивами размером в 9 символов, полезно записать следующую инструкцию определения нового типа:

```
typedef char words[9];
```

Теперь рассмотрим объявление нашего класса:

```
classjukebox
{
protected:
   char directory[30], filename[40];
   int  wordnumber;
   words list[40];
   void wharfile(words name);
public:
   int howmanywords();
   void pronounce(words thisword);
   void pronounce(int whichword);
   void spell(int which, words shelled);
   jukebox(char path[]);
   jukebox();
   int find(words which chone);
};
```

Обратите внимание, что функций-членов с именем pronounce() две. Одна из них получает и озвучивает (с помощью одноименного звукового файла) слово в виде символьного массива. Другая получает целое значение, обозначающее номер слова в списке. Функция howmanywords() просто сообщает количество слов в списке.

Функция spell() в качестве первого аргумента получает номер слова в списке, затем по этому номеру извлекает из списка символьный массив, содержащий данное слово, и копирует его во второй аргумент.

Как видите, по заданному целому значению, например k, учитель может произнести слово, стоящее под этим номером в списке, а также произнести то же слово с помощью функций-членов классаjukebox.

Конструкторы

Кроме того, в классе jukebox имеются два конструктора для инициализации объектов. Аргументом одного из них является символьный массив, содержащий путь к каталогу, в котором записаны звуковые файлы. Например, c:\\franca\\sounds\\ (помните, что в строке символов обратная косая черта заменяется двойной обратной косой чертой). Когда объект классаjukebox пытается озвучить слово, эта строка является префиксом имени соответствующего файла. Поскольку в классе teacher объявляется объект класса jukebox без параметров, необходим и соответствующий конструктор без параметров. При отсутствии такого конструктора объект классаjukebox создать не удастся.



СОВЕТ Если звуковые файлы находятся в подкаталоге sounds текущего каталога, то путь к ним обозначается sounds\\.

Функция whatfile() собирает вместе путь к каталогу, файл и дополнительные символы, чтобы получился полный путь к файлу. Такой полный путь хранится в символьном массиве filename. Этой функции нет в списке открытых членов класса, поскольку она должна быть доступной только для остальных функций-членов.

Чтобы объекты производных классов имели доступ к членам класса jukebox, эти члены объявлены не закрытыми, а защищенными.

Объявление классов

Объекты класса teacher должны уметь писать, озвучивать и проверять каждое слово из списка. Обычному учителю еще нужны часы, а также место для записи слов и количества набранных баллов. Поэтому можно рассмотреть возможность создания объекта класса blackboard (доска), на котором объект класса teacher мог бы записывать все эти вещи. Однако тем же целям могут служить знакомые нам объекты класса Box.

В качестве данных-членов класса teacher объявляются три информационные рамки: yourword (для написания слова), plus (для вывода числа правильных ответов)

и minus (для вывода числа неправильных ответов). Также необходимо объявить объект класса j ukebox, чтобы объекты класса teacher могли обрабатывать слова.

Объявление класса teacher

Ниже представлено объявление класса teacher:

```
class teacher
{
   bank FistNational;
   Box yourword;
   Clock myclock;
   Box plus, minus;
   jukebox speaker;
   int right, wrong;
publ ic;
   void playlist();
   voidtest();
   teacher(julebox &neon);
};
```

Конструктору класса teacher в качестве аргумента передается объект класса jukebox. Это значит, что учителю для работы нужна звуковая библиотека! Почему этот аргумент передается по ссылке? Просто чтобы избежать ненужного копирования.

А что это за объявление объекта класса bank (банк)? Этот объект должен помочь ученикам видеть результаты своих ответов. Я решил для иллюстрации процесса показывать зеленые или красные монетки. Поскольку аналогичные объекты могут потребоваться в других обучающих программах, я решил, что использование здесь объектов класса bank будет полезным. Объект класса bank просто выстраивает стопку зеленых (правильный ответ) или красных (неправильный ответ) монет. Число соответствующих ответов хранится в целых переменных right и wrong.

Объявление класса bank

Ниже представлено объявление класса bank:

```
class bank
{
protected:
   Circle greencoin, redcoin;
   float xgreen, xred;
   float nextygreen, nextyred;
```

```
public:
   void plus();
   void minus();
   bank();
};
```

Функция plus() добавляет зеленую монету, а minus() — красную. Конструктор необходим для создания монет и определения их начального положения при появлении объекта.

Чтобы лучше адаптировать класс bank к многократному использованию, обозначим место, куда будут складываться монеты. Использование параметров по умолчанию позволит вам при желании опускать координаты.

После этих изменений прототип конструктора класса bank будет выглядеть следующим образом:

```
bank(float x=50, float y=400);
```

После написания функций-членов основная функция программы становится очевидной:

```
void mainprog()
{
   jukebox speaker("sounds\\");
   teacher MrRoberts(speaker);
   MrRoberts.playlist();
   MrRoberts.test();
   speaker.pronounce("allright");
   speaker.pronounce("byebye");
}
```

Обратите внимание на объявление объектов классов jukebox и teacher. Объекту класса jukebox сообщается путь, по которому можно найти звуковые файлы.

Единственным аргументом объекта класса teacher является только что созданный объект класса jukebox. Этот аргумент гарантирует, что учитель при проведении диктанта будет пользоваться именно этой конкретной звуковой библиотекой.

Хранение слов в разных каталогах

Чтобы показать, что диктант окончен, необходимо озвучить слова *allright* и *bye-bye*. Ясно, что эти слова должны в числовой форме храниться в том же каталоге, в котором получает слова объект speaker класса jukebox. А что, если хранить эти слова в другом месте? Б конце концов, можно проводить диктанты с разными списками слов, и было бы разумно хранить каждый список в отдельном каталоге. Так стоит ли иметь в каждом таком каталоге по копии значений allright и bye-bye?

Конечно нет! Допустим, вы хотите тестировать слова из каталога c:\franca\test1\, а приветствия извлекать из каталога c:\franca\sounds1\. Что нужно сделать для этого?

Нужно лишь создать две звуковые библиотеки (два объекта класса jukebox): одну для тестируемых слов, а другую для приветствий:

```
jukebox speaker("c:\\franca\\test1\\");
jukebox greeting("c:\\franca\\sound1\\");
```

Звуковая библиотека speaker будет, как и раньше, вызываться объектом класса teacher, а в последней паре инструкций место объекта speaker займет объект greeting, и тогда наш учитель произнесет прощальные слова *allright* и *bye-bye*.



ПРИМЕЧАНИЕ В рассматриваемой здесь реализации игрушки слово *allright* учитель произносит еще и при правильном ответе. При этом используется та же звуковая библиотека и предполагается, что это слово хранится в том же каталоге, что и остальные слова.

Определение классов

Теперь, когда известно, какие объекты будут использоваться, а в объявлениях классов заданы все необходимые функции, можно перейти к их определению для каждого класса. На данном этапе мы по-прежнему будем загружать список слов для звуковой библиотеки вручную.

Определение класса jukebox

Ниже представлено определение класса jukebox:

```
jukebox::jukebox()
{
   wordnumber=0;
   directory[0]=0;
}
jukebox::jukebox(char path[])
{
   fstream wordlist;
   if(strlen(directory)>30)directory[0]=0;
   strcpy(directory, path); // Каталог
   strcpy(filename, directory);
   strcat(filename, "list");
   strcat(filename, ".txt");
   wordlist.open(filename, ios::in | ios::nocreate); //Имя файла
```

```
   if(fileopen(wordlist)==0)
   {   // Проверка доступности файла:
      Box errormsg("Error:");
      errormsg.say("No List!");
      exit(0);
   }
   for(int k=0; k<100; k++)
   {   // Считывание слов:
      if(wordlist.eof()) break;
      wordlist>>list[k];
   }
   wordnumber=k;                                    // Число слов
}
int jukebox::howmanywords()
{
   return wordnumber;
}
void jukebox::spell(int which, words, spelled)
{
   if(which>=wordnumber)
   {
      strcpy(spelled,"");                           // Ограничение размера!
      return;
   }
   strcpy(spelled, list[which]);                    // Копирование в переменную
spelled
}
void jukebox::whatfile(words list)
{
   strcpy(filename, directory);                     // Префикс
   strcat(filename, list);                          // Имя
   strcat(filename, ".wav");                        // Расширение
}
void jukebox::pronounce(intwhichword)
{
   whatfile(list[whichword]);
   sound(filename);
}
void jukebox::pronounce(words thisword)
{
   whatfile(thisword);
   sound(filename);
}
```



ПРИМЕЧАНИЕ Вы должны выбрать компилятор и поместить непосредственно в самое начало программы либо директиву tfdefine Microsoft, либо директиву fldefine Borland (но не обе вместе), чтобы можно было проверить правильность открытия файла.

Определение класса bank

Ниже представлено определение класса bank:

```
bank: :bank(float x, float y)
{
   xgreen=x;
   xred=x+50;
   nextygreen=y;
   nextyred=y;
   greencoin.resize(20, 40);
   greencoin.color(2, 7);
   redcoin.resize(20, 40);
   redcoin.color(1,7);
}
void bank::plus()
{
   greencoin.place(xgreen, nextygreen);
   greencoin.show();
   nextygreen=nextygreen-10;
}
void bank::minus()
{
   redcoin.place(xred, nextyred);
   redcoin.show();
   nextyred=nextyred-10;
}
```

Определение класса teacher

И наконец, наше последнее определение — определение класса teacher:

```
teacher: :teacher(jukebox &neon)
{
   speaker=neon;                    // Копирование звуковой библиотеки
                                    // для объекта класса teacher
   right=wrong=0;
   plus.label("Right:");
```

```
   plus.say(right);
   minus.label("Wrong:");
   minus.say(wrong);
}
void teacher::playlist()
{
   words oneword;
   intn=speaker.howmanywords();
   for(int k=0; k<n; k++);
   {
      speaker.spell(k, oneword);
      yourword.say(oneword);
      speaker.pronounce(k);
      myclock.wait(2);
      yourword.erase();
   }
}
void teacher::test()
{
   char spelling[20];                    // Слово ученика
   char correct[20];                     // Правильное слово
   int j;
   int count=2*speaker.howmanywords();   // Число вопросов
   for(int k=1; k<=cout; k++)
   {
      j=rand()%speaker.howmanywords();
      strcpy(spelling, "");              // Очистка экрана
      speaker.pronounce(j);
      aslwords(spelling, 9, "Please spell:");
      speaker.spell(j, correct);
      if(stricmp(spelling, correct)==0)
      {
         speaker.pronounce("allright");
         plus.say(++right);
         FistNational.plus();
      }
      else
      {
         yesno("Wrong! Will you study harder?");
         minus.say(++wrong);
         FistNational.(minus);
      }
   }
}
```

Что нового мы узнали?

В этом уроке мы научились

- Создавать объектно-ориентированные приложения.
- Многократно использовать фрагменты программного кода.
- Понимать некоторые дополнительные аспекты разработки приложений.
- Разрабатывать простое объектно-ориентированное приложение.



Дополнительные возможности

- Дополнительные возможности C++
- Дополнительные возможности программирования

Основная цель книги — научить вас, как решать проблемы с помощью компьютера и языка программирования C++. Далее, по мере работы, вы поймете, что придумывать алгоритмы гораздо важнее, чем заучивать наизусть программные рецепты.

Те концепции, которые вы освоите, разрабатывая алгоритмы, помогут вам даже в том случае, если вы будете программировать на другом языке.

Чтобы облегчить процесс обучения, некоторые особенности C++ не вошли в этот начальный курс. По мере развития своих навыков, вы, возможно, решите обратиться к более полному руководству или будете осваивать самые сложные элементы языка самостоятельно[1].

На этом последнем уроке мы кратко опишем те дополнительные особенности C++, которые остались за рамками книги. Очевидно, что одна, даже самая совершенная книга не может охватить все аспекты компьютерного программирования, поэтому здесь мы остановимся на том, что могло бы вам пригодиться в будущем.

Дополнительные возможности C++

Эта книга не охватывает всех возможностей языка программирования C++. Некоторые из них были намеренно опущены. В этом разделе мы познакомимся с некоторыми из них.

Указатели

Указатель (pointer) — это переменная, в которой хранится адрес оперативной памяти компьютера другой переменной. Для указателей в C++ есть несколько приложений. С некоторыми из них мы уже встречались при работе с массивами (в которых фактически используются указатели) и при передаче параметров по ссылке. В языке C передача параметра по ссылке невозможна, поэтому именно в этом языке программирования умение обращаться с указателями исключительно важно.

Включение в программу указателей требует особого внимания и может привести к неудобочитаемому коду.

---

[1] В качестве такого руководства может послужить книга Дж. Элджера «C++: библиотека программиста» (издательство «Питер»).



ПРИМЕЧАНИЕ Интересное приложение указателей можно найти в реализации класса Stage из заголовочного файла franca.h. Объектом класса Stage является экранный объект (производный от класса ScreenObj), содержащий массив указателей на экранные объекты. Каждый раз, когда очередной экранный объект вставляется в объект Stage, адрес этого нового объекта копируется в массив. Вызов любой функции-члена класса Stage заставляет все объекты в массиве поочередно вызывать эту функцию-член. Например, вызов функции-члена show() просто проходит через весь массив, где каждый из объектов тоже вызывает функцию-член show(). Таким образом, объекты поочередно появляются на экране. Аналогичная ситуация имеет место с другими функциями-членами.

Комбинированные операторы

Существует специальная группа операторов присваивания, которые можно использовать для изменения значения переменной. Вот они:

```
+=
-=
*=
/=
%=
```

Например, рассмотрим инструкцию:

```
x+=5;
```

Эта инструкция полностью эквивалентна следующей:

```
x=x+5;
```

Эти операторы позволяют несколько сократить объем вводимого кода. Кроме этого, никакой ощутимой выгоды от них нет, но читабельность вашей программы может пострадать. Тем не менее это очень популярные конструкции, встречающиеся во многих программах, и запомнить их довольно легко.

Перегрузка операторов

Одной из наиболее интересных возможностей C++, которая оказалась за рамками данной книги, является *перегрузка операторов* (operator overloading). В C++ можно определить режим работы любого оператора. Например, в C++ оператор + определен только для операций над целыми и числами с плавающей точкой. Выполнение операции сложения с переменными любых других типов или классов без перегрузки этого оператора невозможно.

Как вы знаете, в C++ можно определять собственные классы: строки, комплексные числа, координаты, матрицы или даже экранные объекты. В любом из этих классов можно определить функцию-член operator+(), описывающую действие оператора + в случае, когда один (или оба) из операндов является объектом нового класса. Например, можно выполнять операции со строками. Следующая инструкция будет иметь смысл, если вы перегрузите операторы == и = для выполнения операций с объектами определенного вами строкового класса:

```
if (stringA == stringB) stringA=a copy;
```

Оператор-функции

Благодаря заголовочному файлу franca.h, вызвав функцию-член insert(), можно вставить экранный объект внутрь объекта класса Stage. Например:

```
Stage sold ier;
soldier.insert(rightarm);
soldier.insert(leftarm);
```

Но поскольку в классе Stage также определена функция-член operator<<(ScreenObj &объект), последние две инструкции можно выполнить с помощью оператора <<:

```
soldier<<rightarm;
soldier<<leftarm;
```

Функция-член, которая определяет действие оператора, называется *оператор-функцией* (operator function). В нашем случае оператор-функция описывает, что нужно делать, если оператор << расположен между объектом класса Stage и объектом класса ScreenObj, который является параметром оператор-функции. В нашем случае действие оператор-функции сводится к вызову функции-члена insert().

Шаблоны

Другим полезным элементом C++ являются *шаблоны* (templates). Точно так же, как использование аргументов расширяет возможности функций, так и использование шаблонов расширяет возможности классов.

Простое приложение шаблонов

В нашем знакомом классе textfile имелось ограничение на размер поля идентификационного номера. Это должна была быть строка размером не более, чем 40 символов. Поле идентификационного номера объявлялось как часть класса:

```
class textfile
{
   ...
```

```
publ ic:
   char id[40];
   ...
```

Чтобы иметь возможность определять размер поля идентификационного номера для каждой программы, можно воспользоваться шаблоном:

```
template<int size>
class textfile
{
   ...
   char id[size];
   ....
```

Затем в своей программе нужно объявить объект класса textfile необходимого размера:

```
textfile myfile<65>;
```

Эффект будет тот же, как если бы вы для своих классов объявили размер поля идентификационного номера с помощью константы 65.

Возможности шаблонов не ограничиваются приведенным здесь простым примером. Изменить можно не только значение числа, но и тип или класс данных, с которыми вы работаете. Таким образом, можно иметь идентификационный номер любого типа, а не только типа char:

```
...
anytype id;
```

Тем не менее помните, что возможности шаблонов ограничиваются только заменой их содержимого на то, что вы предлагаете в объявлении класса. Поскольку символьные массивы сравниваются с помощью функции strcmp(), а значения с помощью операторов отношения, кое-какая работа вам все равно остается.

Шаблоны являются мощнейшим инструментом создания многократно используемого программного обеспечения, поскольку позволяют определять так называемые *родовые классы* (generic classes), то есть классы в максимально общем виде, а затем адаптировать их для конкретных целей.

Дружественные функции и классы

Дружественные функции (friend functions) и *дружественные классы* (friend classes), насколько я знаю, имеются только в языке C++. Друзья — это особые члены класса. При объявлении класса A можно указать (с помощью ключевого слова friend), что класс B является дружественным. Теперь, когда класс B становится дружественным для класса A, все его функции-члены получают доступ к закрытым и защищенным данным-членам класса A. Аналогичная ситуация возникает, когда в объявлении класса задается дружественная функция.

Множественное наследование

Можно создать производный класс от более чем одного базового класса. В этом случае возникает ситуация *множественного наследования* (multiple inheritance), поскольку производным классом наследуются данные-члены и функции-члены всех базовых классов.

Выделение памяти

Б этой книге мы рассматривали только один механизм выделения компьютерной памяти для записи в нее значений наших переменных и объектов. Это было, так называемое, автоматическое выделение памяти. При использовании этого механизма создание переменной и выделение для нее памяти происходит при объявлении переменной. Когда переменная выходит из области видимости, ее содержимое удаляется и соответствующее место в памяти освобождается для других переменных.

Существуют и другие способы выделения памяти. Например, статические переменные продолжают существовать (занимать область памяти), даже выходя из области видимости.

Кроме того, с помощью указателей память для переменной можно выделять динамически, то есть в процессе выполнения программы. Предположим, вам необходим массив, но вы заранее не знаете, сколько элементов он будет содержать. Вместо выделения памяти под максимально возможное число элементов массива можно запустить программу, и она сама определит количество элементов массива, а затем затребует у компьютера выделения необходимого для них места в памяти.

Логические операторы

Для выполнения логических операций в C++ имеется несколько специальных операторов. Это операторы конъюнкции &&, дизъюнкции || и отрицания !.

Возможности программирования

Нельзя научиться кататься на велосипеде, прочитав техническое описание велосипеда. Если вы будете просто изучать все атрибуты языка программирования, великим программистом вы все равно не станете. Понимание того, как нужно организовать свои данные, тщательный выбор возможных алгоритмов и развитие необходимых навыков работы — вот те ключевые элементы, которые сделают из вас хорошего программиста.

Больше работайте с компьютером! Будьте любопытны, экспериментируйте!

Инженерные принципы в программировании

Многие считают, что разработка программного обеспечения — это в основном научная или инженерная работа. Другие говорят, что в этом также есть *элементы* искусства! Независимо от вашей собственной точки зрения, необходимо согласиться, что разработка программного обеспечения всегда требует выполнения некоторых требований инженерного характера.

Важно, чтобы при разработке своих программ вы учли все требования по срокам, бюджету и производительности. Нет смысла разрабатывать самое лучшее в мире программное обеспечение, если оно в нужные сроки окажется недоступным или неоправданно дорогим!

Есть несколько принципов, позволяющих сделать процесс разработки программ наиболее эффективным. Чуть сложнее для понимания проблема обслуживания программного обеспечения. Держу пари, вы предпочли бы разрабатывать собственные программы, чем исправлять ошибки и дорабатывать чужие. Я прав?

И тем не менее вы, как программист, можете практически все свое время тратить не на разработку, а на обслуживание программ. И не только потому, что они содержат ошибки, но и для того, чтобы адаптировать их к текущей ситуации — меняются требования рынка, правила игры, возникают новые проблемы...

Чем легче обслуживать программное обеспечение, тем меньше затраты на обслуживание. Но как научиться разрабатывать такие программы?

Непростой вопрос. Как вы уже могли заметить, удобочитаемый код, многократное использование программных компонентов и хорошая документация решают многое. Чтобы максимально упростить обслуживание свих программ, при их разработке старайтесь предвидеть все возможные модификации, которые могут потребоваться в будущем.

Что нового мы узнали?

В этом уроке мы научились

- Понимать, что в C++ мы еще многого не знаем.
- Понимать, что мы еще многого не знаем в программировании

Алфавитный указатель

A

abstract class, 486
American Standart Code for Information Interchange (ASCII), 410
ancestor class, 357
argument, 89, 124
array, 383
ASCII, 410

B

base class, 351
binding, 487
bug, 89

C

cast, 278
child class, 357
class, 330
class declaration, 335
class definition, 335
compound statement, 175
conditional expression, 207
conditional operator, 177
constructor, 339

D

declaration, 112
default constructor, 348
derived class, 351
descendent class, 357
destructor, 474

E-F

enumerated constant, 271
expression, 167
field, 418
File Manager, 34
flowchart, 173
formatting, 445
friend class, 513
friend function, 513
function, 119
  body, 163
  call, 164
  definition, 162
  header, 162
  overloading, 163
  prototype, 164

G-I

generic class, 513
header file, 131
index, 384
infinite loop, 174
inheritance, 352, 357
inline function, 157
inner loop, 186
input, 144
instance, 355

L

labeled constant, 271
late binding, 487
literal constant, 271
loop, 173
loop invariant, 202

M

matching, 480
member function, 87, 330
message, 87, 330
multiple inheritance, 514

N-0

nested loop, 186
null-terminated string, 411
object, 86, 330
Object Oriented Programming (OOP), 19, 493
OOP, 19, 493
operator function, 512
operator overloading, 511
outer loop, 186
output, 146

P

parameter, 124
parent class, 357
pixel, 283
pointer, 510
polymorphism, 358, 362
private, 338
procedure, 119
protected, 338
prototype, 336
public, 338

Q-R

qualification, 337
record, 418
recursion, 249
reference, 129
relational expression, 207
relational operator, 177
return, 153

S

scope, 132
statement, 87
string, 410
structure, 418

T

template, 512
type matching, 163

U-W

user interface, 219
value, 129
virtual function, 364
void, 153
Windows Explorer, 34

А

абстрактный класс, 486
алгоритм, 91
  рекурсивный, 255
Американский стандартный код для обмена информацией, 410
аргумент, 89, 124
аргумент по умолчанию, 125

Б

базовый класс, 351
беззнаковые целые, 269
бесконечный цикл, 174
библиотека
  math.h, 279
блок-схема, 173

В

ввод информации, 144
ввод потоковый, 443
виртуальная функция, 364
встраиваемая функция, 157
вывод информации, 146
вывод потоковый, 442
выделение памяти, 514

вызов функции, 164
выражение, 167
выражение отношения, 207

Г-Д

глобальный объект, 132
данные-члены, 332
двумерный массив, 392
деструктор, 474, 475
директива, 131
  #define, 468
  #endif, 469
  tfifdef, 469
директивы препроцессора, 84, 468
дочерний класс, 357
дружественная функция, 513
дружественный класс, 513

Ж-З

жучок, 89
заголовок функции, 162
заголовочный файл, 130, 131
  iomanip.h, 447
  iostream.h, 441
  string.h, 413
закрытый член класса, 338
запись структуры, 418
защищенный член класса, 338
значение, 129

И

идентификатор, 111
иерархия каталогов, 29
инвариант цикла, 202
индекс, 384
инициализация, 141
  массива, 394
  объекта, 340
  строки, 412
инкапсуляция, 339
инструкция, 87
  break, 239
  continue, 240
  if, 223, 236
инструкция (*продолжение*)
  if...else, 229
  return, 278
  switch, 243, 245
  typedef, 397
исполняемый файл, 25

К

каталог, 28
класс, 85, 330
  bank, 502, 506
  catalog, 426, 486
  jukebox, 500, 504
  payfile, 481
  saleterm, 428
  satellite, 375
  ScreenObj, 321
  Stage, 319, 321
  teacher, 502, 506
  terminal, 369, 428
  textfile, 471
  базовый, 351
  дочерний, 357
  дружественный, 513
  объявление, 335
  определение, 335, 337
  потомок, 357
  предок, 357
  производный, 351
  родительский, 357
  родовой, 513
  экземпляр, 355
ключевое слово
  class, 336
  const, 272
  return, 153
  void, 153
комбинированный оператор, 511
комментарии, 90, 113
компилятор, 25
компоновка программ, 26
конкатенация строк, 415
константа
  буквальная, 271
  именованная, 271

константа *(продолжение)*
перечислимая, 271
целая, 271
константы, 271
конструктор, 339
конструктор по умолчанию, 348
копирование строк, 414
короткие целые, 270

Л-М

локальный объект, 132
манипулятор
endl, 443, 447
resetiosflags, 449
setfill, 449
setiosflags, 449
setprecision, 448
setw, 448
манипуляторы ввода/вывода, 447
массив, 383
двумерный, 392
инициализация, 394
массивов, 392
сортировка, 403
строковый, 412
числовой, 392
машинный язык, 25
множественное наследование, 514

Н

наследование, 352, 357
множественное, 514
неявная ссылка, 334

О

область видимости, 132
объект, 86, 330
cm, 443
cout, 440, 442
Grid, 297
глобальный, 132
локальный, 132
cm, 441
объектно-ориентированное программирование, 19, 493
объектный файл, 25
объявление, 112
класса, 335
файла, 460
оканчивающаяся нулем строка, 411
оператор
sizeof(), 415
комбинированный, 511
логический, 235
отношения, 234
перегрузка, 511
оператор отношения, 177
оператор-функция, 512
определение класса, 335, 337, 340
определение функции, 162
открытие проекта, 35
открытый член класса, 338
ошибка синтаксическая, 110

П

параметр, 124
перегрузка операторов, 511
перегрузка функций, 163
передача
по значению, 129, 164
по ссылке, 129, 164
переменная, 137
статическая, 514
пиксель, 283
повышение типа, 277
подкаталог, 29
позднее связывание, 487
поле структуры, 418
полиморфизм, 358, 362
полностью определенное имя файла, 29
полный путь к файлу, 29
пользовательский интерфейс, 219
понижение типа, 277
потоковый ввод, 443
потоковый вывод, 442
преобразование типа, 277
препроцессор, 84
прерывание цикла, 239

приведение типа, 278
программирование, 24
проект, 28
  открытие, 35
производный класс, 351
прототип функции, 164
прототип функции-члена, 336
процедура, 119

Р

разработка программы, 25
расширение, 28
режим доступа к файлу, 461
рекурсивная
  сортировка, 405
  функция, 250
рекурсия, 249
родительский класс, 357
родовой класс, 513

С

связывание, 487
синтаксическая ошибка, 110
создание каталога, 34
сообщение, 87, 113, 330
соответствие
  записей структуры, 480
  типов, 163
сортировка
  массива, 403
  рекурсивная, 405
составная инструкция, 175
сравнение символов, 409
сравнение строк, 414
ссылка, 129
  неявная, 334
строка, 410
строковые функции, 413
строковый массив, 412
структура, 418

Т

текущий каталог, 29
тело функции, 163
тип
  double, 270
  float, 270
  int, 268
  long, 268
тип данных, 268

У

указатель, 510
унарный оператор, 235
условное выражение, 207
условный оператор, 177
уточнение имени функции, 337

Ф

файл, 28
  исполняемый, 25
  объектный, 25
  текстовый, 460
флаг
  fixed, 453
  scientific, 453
  showpoint, 453
  skipws, 450
форматирование, 445
функция, 119
  absize(), 284
  acos(), 279
  angle(), 378
  asin(), 279
  ask(), 454
  askwords(), 412, 454
  atan(), 279
  atof(), 417
  atoi(), 416
  ceil(), 279
  center(), 376, 377
  close(), 462
  color(), 284
  cos(), 279
  display(), 466, 467, 476
  dist(), 314, 378
  draw(), 303, 305
  drawf(), 305
  eof(), 462

функция *(продолжение)*
erase(), 284
exp(), 279
fabs(), 279
fileopenQ, 470
floor(), 279
height(), 311, 314
howmanywords(), 501
input(), 476
items(), 371
itoa(), 417
label(), 284
log(), 279
log10(), 279
main(), 85
mainprog(), 85
minus(), 503
open(), 461
operate(), 370
origin(), 296
place(), 284, 360
plus(), 503
rand(), 279
read(), 476
readarray(), 445
recsort(), 405
reset(), 336
resize(), 284
run(), 360, 364
say(), 284
scale(), 296
show(), 284, 321, 444, 447
sin(), 279
sort(), 403
speed(), 378
spell(), 501
sqr(), 279
sqrt(), 279
stepleft(), 359
stepright(), 359
strcat(), 415
strcmp(), 414
strcpy(), 414
stricmp(), 414
strlen(), 415
tan(), 279
функция *(продолжение)*
time(), 336
timeGetTime(), 333
wait(), 336
whatfile(), 501
winmain(), 85
write(), 476
yesno(), 454
виртуальная, 364, 486
дружественная, 513
рекурсивная, 250
строковая, 413
функция-член, 87, 113, 130, 330

Ц

цикл, 173
do...while, 177
for, 178
while, 173
вложенный, 186
внешний, 186
внутренний, 186

Ч

числовой массив, 392
член класса
закрытый, 338
защищенный, 338
открытый, 338

Ш-Я

шаблон, 512
экземпляр класса, 355
язык программирования, 25

КЛУБ ПРОФЕССИОНАЛ

В 1997 году по инициативе генерального директора **Издательского дома «Питер»** Валерия Степанова и при поддержке деловых кругов города в Санкт-Петербурге был основан **«Книжный клуб Профессионал».** Он собрал под флагом клуба профессионалов своего дела, которых объединяет постоянная тяга к знаниям и любовь к книгам. Членами клуба являются лучшие студенты и известные практики из разных сфер деятельности, которые хотят стать или уже стали профессионалами в той или иной области,

Как и все развивающиеся проекты, с течением времени книжный клуб вырос в **«Клуб Профессионал».** Идею клуба сегодня формируют три основные «клубные» функции:

- неформальное общение и совместный досуг интересных людей;
- участие в подготовке специалистов высокого класса (семинары, пакеты книг по специальной литературе);
- формирование и высказывание мнений современного профессионала (при встречах и на страницах журнала).

КАК ВСТУПИТЬ В КЛУБ?

Для вступления в **«Клуб Профессионал»** вам необходимо:

- ознакомиться с правилами вступления в **«Клуб Профессионал»** на страницах журнала или на сайте www.piter.com;
- выразить свое желание вступить в **«Клуб Профессионал»** по электронной почте **postbook@piter.com** или по тел. (812) 1 **03-73-74;**
- заказать книги на сумму не менее 500 рублей в течение любого времени или приобрести комплект «Библиотека **профессионала».**

«БИБЛИОТЕКА ПРОФЕССИОНАЛА»

Мы предлагаем вам получить все необходимые знания, подписавшись **на «Библиотеку профессионала».** Она для тех, кто экономит не только время, но и деньги. Покупая комплект - книжную **полку «Библиотека** профессионала», вы получаете:

- **скидку** 1 5% от розничной цены издания, без учета почтовых расходов;
- при покупке двух или более комплектов - дополнительную **скидку 3%;**
- членство в **«Клубе Профессионал»;**
- подарок - журнал **«Клуб Профессионал».**

Закажите бесплатный журнал «Клуб Профессионал».

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®

www.piter.com

ПИТЕР®

Нет времени

ходить по магазинам?

наберите:

www.piter.com

Здесь вы найдете:

Все книги издательства сразу
Новые книги — в момент выхода из типографии
Информацию о книге - - отзывы, рецензии, отрывки
Старые книги - - в библиотеке и на CD

И, наконец, вы нигде не купите наши книги дешевле!

КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- по телефону: (812) 103-73-74;
- по электронному адресу: **postbook@piter.com;**
- на нашем сервере: **www.piter.com;**
- по почте: 197198, **Санкт-Петербург,** а/я 619 **ЗАО «Питер Пост».**

ВЫ МОЖЕТЕ ВЫБРАТЬ ОДИН ИЗ ДВУХ СПОСОБОВ ДОСТАВКИ И ОПЛАТЫ ИЗДАНИЙ:

- Наложенным платежом с оплатой заказа при получении посылки на ближайшем почтовом отделении. Цены на издания приведены ориентировочно и включают в себя стоимость пересылки по почте (но **без учета авиатарифа).** Книги будут высланы нашей службой **«Книга-почтой»** в течение двух недель после получения заказа или выхода книги из печати.
- Оплата наличными при курьерской доставке **(для жителей Москвы и Санкт-Петербурга).** Курьер доставит заказ по указанному адресу в удобное для вас время в течение трех дней.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, код, количество заказываемых экземпляров.

Вы можете заказать бесплатный журнал «Клуб Профессионал».

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

КНИГА-ПОЧТОЙ

СЕРИЯ «САМОУЧИТЕЛЬ»

Пасько В.

САМОУЧИТЕЛЬ РАБОТЫ НА ПЕРСОНАЛЬНОМ КОМПЬЮТЕРЕ. 5-Е ИЗДАНИЕ



Книга содержит всю необходимую информацию для самостоятельного изучения приемов работы на персональном компьютере. Рассказывается об устройстве персональных компьютеров, установке и настройке операционной системы Windows 98, о приемах работы с приложениями пакета Microsoft Office XP (Word 2002, Excel 2002 и PowerPoint 2002). Больше внимание уделено работе в Интернете и системам электронной почты, средствам мультимедиа, антивирусным программам и архиваторам.
Предназначена для начинающих пользователей, студентов и учащихся.

560 с., 17×24, обл.
Код 376
Цена наложенным платежом 197 р.

СЕРИЯ «САМОУЧИТЕЛЬ»

Левин. А

САМОУЧИТЕЛЬ РАБОТЫ НА КОМПЬЮТЕРЕ. 7-Е ИЗДАНИЕ



У вас в руках книга, которая выдержала шесть изданий и выходит в свет в седьмой раз - и опять в обновленном и дополненном виде. Без преувеличения ее можно назвать «национальным бестселлером» - лучшая из книг о компьютерах для новичков.
Автор нашел свою особую интонацию, свой способ подачи материала, которые выделяют эту книгу из целого ряда пособий для начинающих пользователей и позволяют читателю быстро разобраться во всех основных вопросах и при этом получить удовольствие от чтения.
Новичку предлагаются услуги опытного проводника в пути, однажды пройденном самим автором. Вот основные вехи этого пути: Norton Commander, Norton Utilities, команды DOS, архиваторы, антивирусы, текстовый редактор Лексикон, компьютерные игры, MSWindows 95/98/Me, графический итекстовый редакторы, мультимедийные и служебные программы в системе Windows, текстовый процессор Word, электронные таблицы Excel, Интернет и электронная почта.
Новое, седьмое издание дополнено описанием свежих версий всех программ.

656 с., 17x24, обл.
Код 2294
Цена наложенным платежом 178 р.

СЕРИЯ «САМОУЧИТЕЛЬ»

Альбов А., Хайт А.

КОМПЬЮТЕР ДЛЯ НАЧИНАЮЩИХ



В этой книге авторы постарались в увлекательной форме рассказать о том, как при желании и любознательности человек в любом возрасте может с нуля освоить персональный компьютер, стать весьма квалифицированным пользователем. Настоятельно рекомендуем вместе с нашими героями выполнять описанные в самоучителе действия, и помнить, что конкретная программная среда молжет изменяться, а общие принципы работы с приложениями останутся надолго. Погрузитесь вместе с героями книги в увлекательный и очень нужный сегодня мир компьютерных технологий.

272 с., 17×24, обл.
Код 2368
Цена наложенным платежом 137 р.

ПИТЕР®

КНИГА-ПОЧТОЙ



688 с., 17×24, перепл.
Код 263
Цена наложенным платежом 410 р.

СЕРИЯ «ЭНЦИКЛОПЕДИЯ»

Шалин П.

ЭНЦИКЛОПЕДИЯ WINDOWS XP

Целью данного издания стало подробное и детальное рассмотрение основных аспектов установки, настройки и использования новой операционной системы семейства Windows, а именно - русской и английской версий Microsoft Windows XP. Изложенный в книге материал удобно структурирован по разделам, каждый из которых описывает собственный функциональный элемент данной операционной системы, снабжен пояснениями и комментариями. Каждый пользователь Windows сможет найти здесь все, что ему необходимо для эффективной и комфортной работы. Благодаря методичному и последовательному изложению информации, эту книгу можно рассматривать не только как полезный и подробный справочник, но и как учебное пособие, предназначенное для самостоятельного изучения Windows.



480 с., 17x24, обл.
Код 523
Цена наложенным платежом 150 р.

СЕРИЯ «ЭФФЕКТИВНАЯ РАБОТА»

Симонович С., Евсеев Г.

ЭФФЕКТИВНАЯ РАБОТА: ПОЗНАЙ СВОЙ КОМПЬЮТЕР

В книге рассмотрены особенности аппаратной и программной конфигурации настольных персональных компьютеров платформы IBM PC. Даны рекомендации по эффективному подбору компонентов, представлен опыт применения средств тестирования и диагностики, описаны приемы наладки, настройки, модернизации, оптимизации и восстановления компьютерной системы. Написанная простым и понятным языком, она позволит начинающим пользователям глубже понять архитектуру персонального компьютера и принципы его работы, а опытным пользователям достичь максимально возможной производительности компьютерной системы.



320 с., 12,5x20, обл.
Код 3066
Цена наложенным платежом 110 р.

СЕРИЯ «АНАТОМИЯ ПК»

Кутузов М., Преображенский А.

ВЫБОР И МОДЕРНИЗАЦИЯ КОМПЬЮТЕРА. АНАТОМИЯ ПК. 3-Е ИЗДАНИЕ

В книге рассмотрены как вопросы общего характера - проблемы выбора комплектующих, определение оптимальной конфигурации компьютера в зависимости от нужд пользователя, так и конкретные рекомендации по выбору самых актуальных компонентов для компьютера 2003 года. Материал в книге построен таким образом, что читатель без труда разберется во всех тонкостях процесса выбора и покупки наилучших компонентов, даже если ему не приходилось заниматься этим ранее. Особое внимание уделено фирмам-производителям качественного аппаратного обеспечения, чью продукцию можно без опасений использовать в своем компьютере. Заканчивается книга подробнейшим руководством по самостоятельной сборке и тестированию компьютера.

ПИТЕР®

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

СПЕЦИАЛИСТАМ КНИЖНОГО БИЗНЕСА!

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»

предлагают эксклюзивный ассортимент компьютерной, медицинской, психологической, экономической и популярной литературы

РОССИЯ

Москва м. «Калужская», ул, Бутлерова, д. 17б, офис 207, 240; тел./факс (095) 777-54-67; e-mail: sales@piter.msk.ru

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а; тел. (812) 103-73-73, факс (812) 103-73-82; e-mail: sales@piter.com

Воронеж ул. Ленинградская, д. 138; тел. (0732) 51-95-00; e-mail: piter-vrn@vmail.ru; piterv@comch.ru

Екатеринбург ул. 8 Марта, д. 267б; тел./факс (3432) 25-39-94; e-mail: piter-ural@r66.ru

Нижний Новгород ул. Премудрова, д. 31а; тел. (8312) 58-50-15, 58-50-25; e-mail: piter@infonet.nnov.ru

Ростов-на-Дону ул. Калитвинская, д. 17в; тел. (8632) 95-36-31 , (8632) 95-36-32; e-mail: jupiter@rost.ru

Самара ул. Новосадовая, д. 4; тел. (8462)37-06-07; e-mail: piter-volga@sama.ru

УКРАИНА

Харьков ул. Энгельса, д. 29а, офис 610; тел. (0572) 23-75-63, (0572) 28-20-04, (0572) 28-20-05, факс (0572) 14-96-09; e-mail: piter@tender.kharkov.ua

Киев пр. Красных Казаков, д. 6, корп. 1 ; тел./факс (044) 490-35-68, 490-35-69; e-mail: office@piter-press.kiev.ua

БЕЛАРУСЬ

Минск ул. Бобруйская д., 21, офис 3; тел./факс (37517) 226-19-53; e-mail: piter@mail.by

МОЛДОВА

Кишинев «Ауратип-Питер»; ул. Митрополит Варлаам, 65, офис 345; тел. (3732) 22-69-52, факс (3732) 27-24-82; e-mail: lili@auratip.mldnet.com

Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 103-73-73.**
E-mail: grigorjan@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов.
Обращайтесь по телефонам: **Санкт-Петербург — (812) 103-73-72, Москва - (095) 777-54-67.**

Заказ книг для вузов и библиотек: (812) 103-73-73.
Специальное предложение - e-mail: kozin@piter.com

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

УВАЖАЕМЫЕ ГОСПОДА!
КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
ВЫ МОЖЕТЕ ПРИОБРЕСТИ
ОПТОМ И В РОЗНИЦУ
У НАШИХ РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.

Башкортостан

Уфа, «Азия», ул. Зенцова, д. 70 (оптовая продажа),
маг. «Оазис», ул. Чернышевского, д. 88,
тел./факс (3472) 50-39-00.
E-mail: asiaufa@ufanet.ru

Дальний Восток

Владивосток, «Приморский торговый дом книги»,
тел./факс (4232) 23-82-12.
E-mail: bookbase@mail.primorye.ru

Хабаровск, «Мирс»,
тел. (4212) 30-54-47, факс 22-73-30.
E-mail: sale_book@bookmirs.khv.ru

Хабаровск, «Книжный мир»,
тел. (4212) 32-85-51, факс 32-82-50.
E-mail: postmaster@worldbooks.kht.ru

Европейские регионы России

Архангельск, «Дом книги»,
тел. (8182) 65-41 -34, факс 65-41 -34.
E-mail: book@atnet.ru

Калининград, «Вестер»,
тел./факс (0112) 21-56-28, 21 -62-07.
E-mail: nshibkova@vester.ru
http://www.vester.ru

Ростов-на-Дону, ПБОЮЛ Остроменский,
пр. Соколова, д. 73,
тел./факс (8632) 32-18-20.
E-mail: ostrom@don.sitek.net

Северный Кавказ

Ессентуки, «Россы», ул. Октябрьская, 424,
тел./факс (87934) 6-93-09.
E-mail: rossy@kmw.ru

Сибирь

Иркутск, «ПродаЛитЪ»,
тел. (3952) 59-13-70, факс 51-30-70.
E-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Антей-книга»,
тел./факс (3952) 33-42-47.
E-mail: antey@irk.ru

Красноярск, «Книжный мир»,
тел./факс (3912) 27-39-71.
E-mail: book-world@public.krasnet.ru

Нижневартовск, «Дом книги»,
тел. (3466) 23-27-14, факс 23-59-50.
E-mail: book@nvartovsk.wsnet.ru

Новосибирск, «Топ-книга»,
тел. (3832) 36-10-26, факс 36-10-27.
E-mail: office@top-kniga.ru
http://www.top-kniga.ru

Тюмень, «Друг»,
тел./факс (3452) 21-34-82.
E-mail: drug@tyumen.ru

Тюмень, «Фолиант»,
тел. (3452) 27-36-06, факс 27-36-11.
E-mail: foliant@tyumen.ru

Челябинск, ТД «Эврика», ул. Барбюса, д. 61,
тел./факс (3512) 52-49-23.
E-mail:evrika@chel.surnet.ru

Татарстан

Казань, «Таис»,
тел. (8432) 72-34-55, факс 72-27-82.
E-mail: tais@bancorp.ru

Урал

Екатеринбург, магазин № 14,
ул. Челюскинцев, д. 23,
тел./факс (3432) 53-24-90.
E-mail: gvardia@mait.ur.ru

Екатеринбург, «Валео-книга»,
ул. Ключевская, д. 5,
тел./факс (3432) 42-56-00.
E-mail: valeo@etel.ru