ВАЛЕРИЙ ЛАПТЕВ

bhv

C++

ЭКСПРЕСС-КУРС

- Синтаксис и семантика конструкций C++
- Основы стандартной библиотеки шаблонов
- Классы, наследование, шаблоны, исключения
- "Подводные камни" программирования
- Основы программирования в Windows

Валерий Лаптев

C++

ЭКСПРЕСС-КУРС

Санкт-Петербург
«БХВ-Петербург»
2004

УДК 681.3.068+800.92C++
ББК 32.973.26-018.1
Л24

Лаптев В. В.

Л24 C++. Экспресс-курс. — СПб.: БХВ-Петербург, 2004. — 512 с.: ил.

ISBN 5-94157-358-8

Книга представляет собой руководство по программированию на C++, позволяющее быстро освоиться в данном алгоритмическом языке, и включает как необходимый теоретический материал, так и реализации задуманных программ в виде листингов, поясняющих рисунков, таблиц. Начав с изучения основ языка, читатель знакомится с принципами перехода от формального словесного описания задачи к описанию, понятному для ПК и позволяющему решить ее за короткое время, постепенно осваивает все более сложные конструкции, учится сам использовать богатый арсенал C++. Приводятся примеры не только работающих, "отлаженных" программ, но и наиболее вероятных ошибок, возникающих в процессе написания программы и не всегда распознаваемых компилятором. Рассматриваемые встроенные функции, библиотеки дают возможность при правильном подключении уже готовых функций, макросов значительно сократить программный код.

Для начинающих программистов

УДК 681.3.068+800.92C++
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Анатолий Адаменко*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Яковлева*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 02.12.03.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 41,28.
Тираж 5000 экз. Заказ №
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-358-8

© Лаптев В. В., 2004
© Оформление, издательство "БХВ-Петербург", 2004

Содержание

Введение **9**

Кому адресована эта книга 11
Структура книги 11
Используемые программные продукты 12
Благодарности 13

ЧАСТЬ I. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА C++ **15**

Глава 1. Как написать программу на C++ **17**

Проверка условий 22
Оператор-переключатель 27
Повторение вычислений 29
Операторы *break* и *continue* 33
Встроенные типы данных 34
Дробные числа 34
Целые числа 40
Символы и строки 46
Операции присваивания и выражения 49
Преобразование типов 51
Другие операции 52
Печальная действительность 57
"Страшный зверь" по имени *undefined behavior* 59

Глава 2. Функции **61**

Библиотечные функции 62
Математические функции 63
Как написать функцию 66
Определение функции 68
Список параметров и вызов функции 70
Передача параметров по значению 71
Параметры по умолчанию 73

Область видимости и "время жизни" переменных ... 77
Передача параметров по ссылке ... 82
Константные параметры функций ... 84
Побочный эффект ... 85
Перегрузка функций ... 89
Шаблоны функций ... 92
Символы ... 95
Макросы и inline-функции ... 99
Применение препроцессора с пользой ... 101

Глава 3. Группы данных ... 105

Массивы ... 105
Обработка числовых массивов ... 109
Индексы как параметры ... 116
Многомерные массивы ... 119
Строки ... 120
Обработка символьных массивов ... 123
Ввод и вывод массивов символов ... 125
Структуры ... 127
Структуры как параметры ... 129
Шаблоны структур ... 132
Массивы и структуры ... 134
Структуры, функции и шаблоны ... 135
Размеры структур ... 140

Глава 4. Тяжелое наследие C ... 143

Параметры-массивы в форме указателя ... 143
Что такое указатели ... 144
Виды указателей ... 147
Объявление типизированных указателей ... 148
Бестиповые указатели и преобразование типов ... 150
Массивы и указатели ... 151
Указатели как параметры ... 153
Бестиповые указатели как параметры ... 155
Указатели на символы ... 157
Русские буквы ... 158
Библиотека string.h ... 159
Указатели на символы — переменные и константы ... 161
Указатели и динамическая память ... 164
Многомерные динамические массивы ... 166
Многомерные массивы как параметры ... 168
Структуры и указатели ... 169
Структуры с указателями ... 169
Проблемы с указателями ... 173
Указатели на функции как параметры ... 175
Экономия памяти ... 177
Параметры функции *main* ... 179

Глава 5. Стандартная библиотека ... 183

Логический тип данных ... 184
Новые строки ... 184
Строки как параметры ... 189
Числа — прописью ... 191
Контейнеры, итераторы и алгоритмы ... 195
Векторы вместо массивов ... 196
Указатели и контейнеры ... 203
Стандартные алгоритмы и итераторы ... 204
Поиск в контейнере ... 209
Сортировки ... 214
Обработка контейнеров и функциональные объекты ... 215

Глава 6. Ввод и вывод в C++ ... 221

Стандартные потоки в C++ ... 222
Ввод данных ... 223
Ввод строк ... 225
Потоки и файлы ... 227
Каталоги ... 228
Протестируемся ... 231
Состояния потока ... 234
Макет сохранения информации ... 235
Программа тестирования ... 238
Форматирование вывода ... 240
Снова о программе тестирования ... 242
Режимы открытия потоков (файлов) ... 246
Текстовые и двоичные файлы ... 247
Двоичные файлы и прямой доступ ... 249
Шифрование файлов ... 250
Соберем все вместе ... 254
Строковые потоки ... 257

Глава 7. Снова о функциях ... 259

"Левые" функции ... 259
"Левые" функции с указателями ... 262
Функции с переменным числом параметров ... 264
Стандартные средства ... 270
Рекурсивные функции ... 272
Формы рекурсивных функций ... 275
Выполнение рекурсивных функций ... 278
Рекурсия при обработке динамических структур данных ... 284
Односвязный линейный список ... 284
Двоичное дерево ... 286
Параметры в рекурсивных функциях ... 289

ЧАСТЬ II. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ 291

Глава 8. Создание простых типов 293

Перегрузка операций 293
Перегрузка операций для *enum* 295
Снова структуры 300
Конструкторы 306
Конструкторы и параметры функций 311
Конструкторы и преобразование типов 312
Перегрузка операций методами 316
Внешняя или внутренняя? 319
Классы и структуры 322

Глава 9. Динамические классы 327

Массивы с задаваемыми индексами 327
Конструкторы и деструкторы 330
Копирующее присваивание 332
"Разноликие" матрицы 335
Перегрузка индексирования для нецелых аргументов 339
Последовательный контейнер 340
"Интеллектуальные" указатели 343
auto_ptr 345

Глава 10. Исключения — что это такое 347

Принципы обработки исключений 347
Генерация исключений 348
Перехват исключений 349
Функции и исключения 350
Передача информации в блок обработки 352
Спецификация исключений 353
Конструкторы, деструкторы и исключения 354
Стандартные исключения 357

Глава 11. Наследование 361

Простое наследование 361
Открытое наследование 363
Конструкторы и деструкторы в производных классах 365
Закрытое наследование 368
Виртуальность 370
RTTI и *dynamic_cast* 374
Большие программы 376
Переменные, функции и файлы 377
Пространства имен 379
Именованные пространства имен 380
Неименованные пространства имен 382
Использование препроцессора 384

Глава 12. Обобщенное программирование 387

Процедурное обобщенное программирование 388
Полиморфные функции 391
Способы передачи параметров-указателей на функции 393
Указатели на функцию в списках переменной длины 393
Возврат указателя на функцию 398
Функционально-объектное обобщенное программирование 401
Решения стандартной библиотеки 405
Объектно-ориентированное обобщенное программирование 407
Абстрактные классы 409
Метапрограммирование 410

ЧАСТЬ III. КАК НАПИСАТЬ ПРОГРАММУ ДЛЯ WINDOWS 415

Глава 13. Windows — это не просто 417

Венгерская нотация 417
Консольные приложения и Unicode 420
Оконные приложения 425
Создание "пустой" программы 425
Интерфейс программы умножения 430
Создание меню 431
Создание элементов управления 433
Обработка сообщений 436
Стоила ли игра свеч 441

Глава 14. Быстрая разработка приложений 443

Шаг 1 443
Шаг 2 444
Шаг 3 447
Шаг 4 450
Шаг 5 451

Заключение 457

ЧАСТЬ IV. ПРИЛОЖЕНИЯ 459

Приложение 1. Системы фирмы Borland 461

Borland C++ 3.1 461
Запуск IDE и выход 462
Создание и сохранение программы 463
Многофайловые программы 465
Создание и изменение проекта 465
Компиляция, компоновка и выполнение 466
Работа с отладчиком 466
Точки прерывания 466
Выполнение до курсора 467

Пооператорное (пошаговое) выполнение ... 467
Просмотр и изменение переменных ... 467
Справочная система ... 467
Borland C++ 5 ... 470
Запуск IDE ... 470
Создание и выполнение консольных программ ... 470
Продолжение работы с программой ... 474
Создание проекта программы ... 474
Работа с отладчиком ... 476
Справочная система ... 476
Borland C++ Builder 6 ... 479
Запуск IDE ... 479
Создание нового проекта ... 479
Продолжение работы с программой ... 482
Компиляция, компоновка и выполнение ... 482
Работа с отладчиком ... 482
Справочная система ... 483

Приложение 2. Интегрированная среда Microsoft Visual C++ 6 ... 485

Запуск IDE ... 485
Создание и выполнение программ ... 485
Продолжение работы с проектом ... 489
Конфигурация проекта ... 490
Работа с отладчиком ... 490

Приложение 3. Ресурсы C++ в Интернете ... 493

Приложение 4. Список литературы ... 495

Предметный указатель ... 499

Введение

По шутливому замечанию Джеффа Элджера [49], "на рынке продается по крайней мере 2 768 942 книги о C++". И все же вы держите в руках еще одну — 2 768 943-ю книгу о том же самом. Зачем нужна еще одна книга о C++?

Когда я стал преподавать программирование на языке C++ для первокурсников, мне потребовалось выбрать базовый учебник, по которому студенты станут изучать C++ и учиться программировать на этом языке. Перерыв "гору" литературы, я с удивлением констатировал, что нужной мне книги нет. По той или иной причине ни одна из них меня не устроила. Оказалось, что некоторые очень важные свойства языка часто либо вовсе не описаны, либо рассматриваются на элементарном уровне. Наверное, лучшим выбором была бы книга Стенли Липпмана "C++ для начинающих", однако в настоящее время ее просто не найти.

Все книги о C++ можно разделить на несколько категорий: о самом языке C++, об использовании конкретной среды программирования вроде Borland C++ Builder или Visual C++, и о типовых структурах данных и алгоритмах. Однако по собственному опыту знаю, что этого недостаточно, чтобы научиться программировать на C++. В жизни программисту приходится сталкиваться с целым спектром чисто практических проблем, которые по непонятным мне причинам практически ни в одной книге не излагаются.

Во-первых, как я неоднократно ранее убеждался на собственном опыте и еще больше уверился на опыте моих студентов, изучить язык программирования в отрыве от среды программирования нельзя. Можно прекрасно выучить все нюансы C++, но быть совершенно беспомощным при написании конкретной программы в определенной системе. Обучение всегда должно происходить в некоторой среде программирования, да и работать всегда приходится в реальной системе, которая имеет те или иные особенности и недоработки.

Во-вторых, программировать, не обращая внимания на операционную систему и аппаратную платформу, тоже невозможно — хотя бы потому, что реализация встроенных типов данных зависит от аппаратуры. И тут существует множество "подводных камней", на которые постоянно "натыкается" начинающий программист. Ввод/вывод в любом языке программирования всегда зависит от файловой системы ОС. Недаром, чтобы уменьшить эту зависимость, ввод/вывод в C++ полностью вынесен в отдельные библиотеки.

И наконец, проблема русских букв. Естественно, ни одна из книг иностранных авторов, многие из которых просто замечательны и мне очень нравятся, эту проблему не затрагивает. Удивительно, но и в книгах российских авторов тоже нет об этом ни слова! Между тем, это серьезная проблема для начинающего программиста, требующая определенных знаний и навыков.

Такой же значимой проблемой является переход от консольных приложений к оконным. Помню, как мучительно долго я, опытный программист, разбирался в идеологии и особенностях программирования для Windows. Угнетало количество технических деталей и отсутствие четких ориентиров, на что в первую очередь обращать внимание. Тем более такой переход сложен для новичка, несмотря на то, что сейчас существует достаточно много отличных книг, обучающих программированию для Windows.

Однако в экспресс-курсе невозможно изложить все. В процессе работы над книгой мне часто приходилось решать, что оставить, а что выбросить. Должен признать, что отбор материала для изложения оказался большой проблемой — язык C++ очень обширен. Поэтому, может быть, не все читатели найдут в книге то, что хотели найти. Например, в *гл. 4* об указателях я решил не упоминать о функции `malloc` и ее "родственниках" из C, а в *гл. 6* о вводе/выводе ни слова нет о библиотеке stdio.h. Не упоминается и о битовых полях в структуре и совсем мало о битовых операциях. По некоторым темам, только затронутым в книге, можно было бы написать отдельную книгу, в два раза большую по объему (возможно, мне удастся сделать это в будущем). А вот о программировании функций, наоборот, написано значительно больше, чем это обычно принято. Я убежден, что без фундаментальных знаний о функциях невозможно программировать. Довольно много внимания уделено и различным проблемам, возникающим в реальном программировании. Достаточно подробно описана проблема "кириллизации" строк, так важных для начинающих программистов. К сожалению, объем книги не позволял изложить стандартные средства локализации в достаточном объеме, поэтому я решил вообще не касаться этой темы.

В итоге получилась книга, которую вы держите в руках. О ее достоинствах и недостатках — судить вам.

Кому адресована эта книга

Книга представляет собой практическое введение в программирование на C++ в распространенных в России средах фирмы Borland и Visual C++. Книга написана, в первую очередь, для моих студентов — начинающих программистов, которые хотели бы научиться программировать на C++. Желательно (но не обязательно), чтобы имелся опыт программирования (хотя бы учебных программ) на другом языке. Однако и более опытные программисты, надеюсь, найдут в книге немало интересного.

Структура книги

Книга состоит из 4-х частей. В *части I* излагаются основы языка C++. В *гл. 1* описываются основные операторы, встроенные типы данных и операции. Рассматриваются типичные ошибки и проблемы, возникающие, например, при переполнении или неопределенном поведении программы.

Гл. 2 посвящена функциям. Описываются способы передачи параметров по значению и по ссылке, константные параметры, параметры по умолчанию. Уделяется внимание некоторым проблемам, связанным с областью видимости переменных. Рассматривается перегрузка и шаблоны функций, основы использования препроцессора.

В *гл. 3* описаны массивы и структуры, приведены многочисленные примеры обработки массивов, в том числе и символьных. Рассматриваются "взаимоотношения" массивов и структур, разбираются основы шаблонов структур.

Без сомнения самой важной темой C++ являются указатели и динамическая память. Об этом довольно подробно повествует *гл. 4*.

В *гл. 5* излагается элементарное введение в библиотеку стандартных шаблонов. Достаточно много внимания уделено строкам.

В *гл. 6* на примере программирования небольшой автоматизированной системы тестирования описываются основы системы ввода/вывода C++.

Гл. 7 возвращает нас к функциям. Здесь излагаются более сложные вопросы: рекурсия, функции с переменным числом параметров, так называемые "левые" функции.

В *части II* излагаются основы объектно-ориентированного и обобщенного программирования на C++. Объясняется реализация некоторых простых паттернов проектирования. В *гл. 8* подробно разбираются вопросы перегрузки операций, определяется понятие конструктора, реализуется почти законченный класс рациональных чисел.

Гл. 9 посвящена программированию динамических классов. На примере динамических массивов и списков разбираются проблемы, возникающие при

программировании классов с динамическим распределением памяти. Разбирается понятие интеллектуального указателя.

Гл. 10 содержит то, что начинающему программисту необходимо знать об исключениях.

В *гл. 11* рассматривается один из трех краеугольных камней объектно-ориентированного программирования — наследование, и связанные с ним вопросы: виртуальные функции и RTTI. Попутно разбирается организация многомодульных программ.

В *гл. 12* излагаются некоторые особенности STL, рассматривается программирование шаблонов и функциональных объектов.

Часть III показывает переход к программированию для Windows. В *гл. 13* рассматривается использование Win32 API для создания оконных приложений.

В *гл. 14* приведен пример практической разработки приложения с использованием C++ Builder.

В *приложениях (часть IV)* рассмотрены характеристики систем программирования, приведены ссылки для поиска дополнительной информации в Интернете, а также составлен перечень литературных источников, знакомство с которыми расширит ваш кругозор.

Используемые программные продукты

В *приложениях 1, 2* приводятся краткие сведения о нескольких системах. Почти все примеры были реально проверены в лицензионной системе Microsoft Visual C++ 6, в которой поставляемая STL от DinkumWare была заменена на STLport версии 4.5.3. Отдельные примеры для сравнения проверялись в нелицензионной версии Visual C++ 7. Однако, по глубокому убеждению автора, квалифицированный программист не должен испытывать затруднений при работе в нескольких разных системах. Поэтому в качестве альтернативы были использованы столь любимые российскими программистами системы фирмы Borland: Borland C++ 3.1, Borland C++ 5, Borland C++ Builder 6. Выбраны системы Borland, как наиболее известные в России. Кроме того, у всех "борландовских" систем прекрасная система встроенной помощи, тогда как Visual C++ 6 требует отдельной установки MSDN.

Может вызвать некоторое недоумение упоминание системы Borland C++ 3.1. Я использую эту версию по следующим причинам.

- Если мы начинаем работать в Borland C++ 3.1, то отсутствует проблема с русскими буквами — все сообщения выводятся именно в том "русском виде", как программист набрал их в редакторе. Практически все примеры первых трех глав проверены именно в этой системе.

- Borland C++ 3.1 абсолютно совместима с Borland (Turbo) Pascal 7 по внешнему виду, "горячим клавишам", составу подсистем и т. д. Таким образом, "паскалистам", начинающим изучать C++ (а это практически все студенты 1-го или 2-го курса почти всех университетов России), будет значительно проще освоиться с новым языком.
- После версии 3.1 легко перейти к другим, более мощным "борландовским" системам, т. к. состав клавишных команд остается практически неизменным.
- Borland C++ 3.1 до сих пор реально используется в промышленном программировании. Система достаточно мощная — в ней нет только стандартной библиотеки STL и исключений — это появилось позже. Остальное все есть. Мой друг, работающий в Санкт-Петербурге в одной серьезной организации, написал мне в ответ на вопрос об используемых системах программирования: "Мы пишем все на строгом ANSI C... Мы применяем компилятор Борланд 3.1 и 4.5, еще MS 4.1. Проверены. Я лично ЗАПРЕТИЛ применения Борланда 5 и выше, когда исследовал несколько объектных файлов". Да и в форумах **www.rsdn.ru** частенько попадаются вопросы об этой системе.
- На командных чемпионатах мира по программированию среди студентов, ежегодно проводимых под эгидой IBM, по правилам соревнований допускается использование только двух систем: Borland (Turbo) Pascal 7 и Borland C++ 3.1. Поэтому практически во всех университетах России (и мира тоже) обязательно их изучают как первые системы программирования.
- Еще одна причина — систему легко найти, она бесплатно передается из рук в руки, в отличие от более поздних систем.

Операционной системой, естественно, выбрана Windows, т. к. все среды программирования работают только в ней.

Благодарности

В первую очередь хотел бы выразить благодарность сотрудникам издательства "БХВ-Петербург", без самоотверженного труда которых эта книга просто не могла родиться. Спасибо моим студентам, работа с ними натолкнула меня на мысль о необходимости написать эту книгу. Не могу не поблагодарить членов команды RSDN, в общении с которыми я провел много часов, выясняя нюансы тех или иных конструкций C++.

ЧАСТЬ I

Основы программирования на C++

Глава 1. Как написать программу на C++

Глава 2. Функции

Глава 3. Группы данных

Глава 4. Тяжелое наследие C

Глава 5. Стандартная библиотека

Глава 6. Ввод и вывод в C++

Глава 7. Снова о функциях

ГЛАВА 1

Как написать программу на C++

"Поехали!"
Ю. Гагарин

С момента издания книги "Язык программирования C", написанной Брайаном Керниганом и Денисом Ритчи, стало модным начинать любую книгу по программированию с программы "Hello, World!". Мы немного отступим от этой традиции и сразу "возьмем быка за рога": будем использовать компьютер по прямому назначению — для вычислений. Запустим среду Borland C++ 3.1 и в окне редактора наберем программу "Дважды два", текст которой приведен в листинге 1.1.

Листинг 1.1. Программа "Дважды два"

```
//  Программа "Дважды два"
#include <iostream.h>
int main()
{  cout << "2 * 2 =" << 2 * 2 << endl;
   return 0;
}
```

Выполним программу (нажмем клавиши <Ctrl>+<F9>). Если все сделано правильно, то, нажав комбинацию клавиш <Alt>+<F5> и открыв экран пользователя, увидим на экране строку:

```
2 * 2 = 4
```

Разберем программу построчно. Первая строка — это *строчный комментарий*. Такой комментарий всегда начинается с двух символов // и продолжается до конца строки. Начало строчного комментария может быть в любом месте строки, он не обязательно должен начинаться с начала строки. В строчном комментарии, как и в любом другом, можно писать любые символы, доступные на клавиатуре, — транслятор пропустит такую строку.

В C++ есть и *многострочный* комментарий, который появился еще в языке C. Такой комментарий начинается с символов `/*` и заканчивается символами `*/`. Начало и конец комментария могут располагаться на разных строках исходного текста программы, и обычно их пишут на отдельных строках, как показано в следующем примере:

```
/* Это
   многострочный комментарий
*/
```

Внутри многострочного комментария вполне может быть однострочный. Так обычно случается, если программист в процессе работы над программой закомментировал некоторый фрагмент, в котором был написан однострочный комментарий. Однако вложенные многострочные комментарии не допускаются. Такая "дискриминация" возникает потому, что окончанием комментария считается первое встретившееся сочетание символов `*/` — все, что после них, опять считается программой.

Примечание

Так требует стандарт. Однако системы фирмы Borland позволяют установить режим работы, при котором вложенные комментарии допускаются.

Следующая строка — это подключение библиотеки ввода/вывода. В языке C++, как и ранее в языке C, отсутствует встроенная система ввода/вывода, и весь обмен данными выполняется внешними программами. В данном случае эта строка обеспечивает нам работу оператора вывода на экран. Имя iostream.h — это имя включаемого файла, который находится в каталоге интегрированной среды include. В языке C++ многие *расширения* подключаются подобным образом. В последней версии C++ все они составляют *стандартную библиотеку*, в состав которой входят две стандартные библиотеки ввода/вывода.

Строка начинается со знака `#` (решетка). Этот символ используется, чтобы обозначить *директивы* препроцессора. *Препроцессор* — это программа, которая выполняет простую обработку текста для последующей трансляции. Существуют и иные директивы препроцессора, например `define`, и ряд других.

Выполнение программы (консольного приложения), написанной на языке C++, всегда начинается с выполнения *главной* функции `main`. Как и всякая другая функция, функция `main` состоит из заголовка

```
int main()
```

и тела в фигурных скобках.

Заголовок главной функции `main` имеет стандартный вид:

- сначала указывается *тип возвращаемого значения*. В данном случае тип возвращаемого значения — стандартный *целый* тип `int`, это слово является *зарезервированным словом* языка C++ и его нельзя писать по-другому. Язык C++ включает и иные зарезервированные слова, которые мы будем узнавать по мере изучения;
- `main` — обязательное имя главной функции; другое имя указывать нельзя. Несмотря на это, слово `main` не является зарезервированным словом языка C++;
- скобки после имени показывают, что `main` — это именно имя функции.

Все слова в заголовке написаны маленькими английскими буквами. Если мы попробуем использовать в заголовке хотя бы одну большую букву, программа не будет транслироваться. В языке C++ все зарезервированные слова пишутся маленькими буквами.

В теле функции записывается последовательность действий, которые требуется выполнить. Эта последовательность действий записывается с помощью *операторов*, каждый из которых завершается символом `;` (точка с запятой). В нашей программе в теле функции задан *оператор вывода* на экран

```
cout << "2 * 2 =" << 2 * 2 << endl;
```

и *оператор возврата* значения

```
return 0;
```

Английское слово `return` является зарезервированным словом языка C++; его надо писать именно так, как показано, и нельзя использовать никаким другим образом. Число `0`, заданное в операторе, и есть целое возвращаемое значение, которое может быть проверено с помощью команд операционной системы. Обычно значение `0` означает нормальное завершение программы. Это только соглашение, которого стараются придерживаться все программисты в мире. Однако вы можете написать программу, в которой в качестве нормального возвращаемого значения используете значение `1` или `-1`. Тем не менее постарайтесь придерживаться общепринятого соглашения, потому что соглашения (поверьте моему опыту) облегчают программисту жизнь.

Оператор вывода имеет более сложную структуру:

- `cout` — это *системный объект*, обеспечивающий *вывод* результатов из программы во внешнюю среду, в нашем случае — на экран. Данное слово не является зарезервированным словом C++, тем не менее, во избежание возможной путаницы, не рекомендуется использовать `cout` в каком-нибудь другом смысле;
- в нашем операторе вывода задано три аргумента: строковая константа `"2 * 2 ="`, выражение `2 * 2` и манипулятор `endl`. Перед каждым аргумен-

том прописан двойной знак меньше <<. Это одна из операций языка, которая в данном случае означает "вывод в поток".

Строковая константа выводится точно в таком виде, как написана, только без кавычек. При написании программы мы можем внутри кавычек писать любые символы (кроме двойных кавычек, одиночной кавычки-апострофа и символа \ — обратной косой черты), которые есть на клавиатуре. В частности, без ограничений можно использовать русские и английские буквы. Вне кавычек русские буквы можно применять только в комментариях.

Выражение содержит три элемента: две целых константы и знак * (звездочка), который обозначает операцию умножения. Выражение вычисляется, и значение выводится на экран.

Манипулятор `endl` обеспечивает нам перевод курсора экрана на следующую строку.

Теперь мы легко можем написать и выполнить программу, вычисляющую и выводящую на экран полную таблицу умножения на 2 — мы просто скопируем 10 раз оператор вывода, исправив множители в строках и в выражениях. Понятно, что таким же способом мы можем создать программы для вычисления каких угодно таблиц умножения. Однако лучше написать программу таким образом, чтобы ее можно было использовать с различными данными. В самом деле, таблица умножения на 3 отличается от таблицы умножения на 2 только множителем. Этот множитель можно было бы сообщать программе каждый раз при запуске. Для этого в программе (ее текст приведен в листинге 1.2) необходимо объявить переменную и написать оператор ввода.

Листинг 1.2. Таблица умножения с вводом переменной

```
#include <iostream.h>
int main()
{  int k;       //  объявление переменной
   cout << "Введите множитель от 1 до 9: ";
   cin >> k;    //  ввод числа
   cout << k << "* 1 =" << k * 1 << endl;
   cout << k << "* 2 =" << k * 2 << endl;
   cout << k << "* 3 =" << k * 3 << endl;
   cout << k << "* 4 =" << k * 4 << endl;
   cout << k << "* 5 =" << k * 5 << endl;
   cout << k << "* 6 =" << k * 6 << endl;
   cout << k << "* 7 =" << k * 7 << endl;
   cout << k << "* 8 =" << k * 8 << endl;
   cout << k << "* 9 =" << k * 9 << endl;
   cout << k << "* 10 =" << k * 10 << endl;
   return 0;
}
```

В этой программе оператор

```
int k;
```

объявляет целую *переменную* с именем k. Имя переменной называется *идентификатором*. Для написания идентификатора программист может использовать большие и маленькие английские буквы, цифры и знак подчеркивания _. Идентификаторы в языке C++ должны начинаться либо с буквы, либо с подчеркивания. С цифры начинать идентификатор нельзя. Подчеркивание тоже первым писать не рекомендуется, хотя и разрешается. Мы вместо слова "идентификатор" часто будем употреблять слово "имя" — оно короче. В C++ различаются большие и маленькие буквы, поэтому следующие имена считаются различными: ab, Ab, aB, AB. Количество символов в имени язык не ограничивает, но понятно, что бесконечно длинным имя быть не может. В каждой интегрированной среде обычно устанавливается некоторое ограничение длины имен.

Слово int — это тип значений, которые могут храниться в нашей переменной. Как мы помним, int является зарезервированным словом. Каждый тип определяется множеством значений и набором операций, которые применяются к значениям. В любом языке программирования обычно есть некоторый набор *встроенных* типов, каждый из которых определяется своим зарезервированным словом. Язык C++ включает много различных встроенных типов, большинство из которых являются числовыми.

Таким образом, каждая переменная, объявленная в программе, имеет имя и ей приписан тип. В разные моменты времени в переменной могут храниться разные *значения*, однако имя и тип изменить нельзя. В общем случае имена всех переменных должны быть разные, однако в C++ есть исключения из этого правила.

После объявления переменной мы написали оператор для вывода приглашения. После него оператор

```
cin >> k;
```

выполняет *ввод* значений переменной k. Слово cin обозначает системный объект, обеспечивающий нам ввод данных из внешней среды в программу. В данном случае выполняется ввод с клавиатуры. Последней клавишей, которую требуется нажать, должна быть клавиша <Enter>.

Так же, как и cout, слово cin не является зарезервированным словом C++, однако не рекомендуется использовать его в каком-нибудь другом смысле. Мы видим также, что C++ позволяет программисту записывать несколько операторов в одной строке, лишь бы каждый оператор заканчивался точкой с запятой. После последнего оператора в строке можно написать строчный комментарий.

Далее написаны десять однотипных операторов вывода. Если мы запустим программу, то на экране появится сообщение

```
Введите множитель от 1 до 9:
```

и программа остановится, ожидая от нас ввода целого числа. Нажмем цифру 5 и клавишу <Enter>, и увидим на экране следующие десять строк:

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

Каждый из операторов вывода работает так: сначала на экран выводится значение переменной k (в данном случае — число 5), потом без пропуска — соответствующая строковая константа, затем вычисляется соответствующее выражение и выводится его значение.

Проверка условий

Если мы вновь запустим программу (см. листинг 1.2) и введем значение 3, то увидим на экране таблицу умножения на 3. Более того, мы можем ввести любое положительное число, например 147, и получим таблицу умножения на 147. Мы даже можем вычислять таблицы умножения отрицательных чисел. Запустим программу и введем значение -2. Программа правильно вычислит и выведет на экран отрицательную таблицу умножения. Таким образом, программа работает, и достаточно универсальна. Тем не менее изначально требовалось иметь программу, вычисляющую таблицу умножения от 1 до 9. Наша программа делает "лишнюю" работу.

"Корифеи" программирования в один голос утверждают: если программа выполняет не предусмотренную первоначальным заданием работу, то такая программа ошибочна. Таким образом, в программе необходимо проверять значения, которые вводит пользователь, и отвергать ошибочные. Пусть программа в этом случае выводит на экран вежливое предупреждение и заканчивает работу. Напишем это более формальным способом:

```
если 1 ≤ к ≤ 9 то вывести таблицу умножения;
иначе вывести сообщение об ошибке;
```

Чтобы запрограммировать эти действия, нам потребуется *условный* оператор. На языке C++ условный оператор записывается так:

```
if (условие) оператор;
```

или так:

```
if (условие) оператор_1;
else оператор_2;
```

Слова `if` и `else` являются зарезервированными словами. Первая форма оператора `if` работает так: если условие истинно, то выполняется заданный оператор. Затем выполняется следующий оператор после условного. Если же условие ложно, то заданный оператор пропускается и сразу начинает работать следующий после `if` оператор. Вторая форма оператора работает таким образом: если условие истинно, то выполняется заданный `оператор_1`; если условие ложно, то работает заданный `оператор_2`. Затем выполняется оператор, следующий после условного.

Как мы видим, вторая форма оператора практически совпадает с написанной на русском языке *схемой*. Однако нам надо решить две проблемы. Первая проблема состоит в том, что наша фраза "вывести таблицу умножения" означает запись на языке C++ десяти операторов вывода, в то время как в операторе `if` можно указывать только один оператор. Эта проблема решается очень просто — с помощью фигурных скобок. Оператор `if` в этом случае приобретает вид:

```
if  (условие)
{  //    составной оператор
   cout << k << "* 1 =" << k * 1 << endl;
   cout << k << "* 2 =" << k * 2 << endl;
   cout << k << "* 3 =" << k * 3 << endl;
   cout << k << "* 4 =" << k * 4 << endl;
   cout << k << "* 5 =" << k * 5 << endl;
   cout << k << "* 6 =" << k * 6 << endl;
   cout << k << "* 7 =" << k * 7 << endl;
   cout << k << "* 8 =" << k * 8 << endl;
   cout << k << "* 9 =" << k * 9 << endl;
   cout << k << "* 10 =" << k * 10 << endl;
}
else cout << "Неправильное значение множителя!" << endl;
```

Несколько операторов, заключенных в фигурные скобки, называются *составным оператором*. В тексте программы составной оператор может стоять везде, где может быть указан простой оператор. Если нам потребуется, мы можем задать составной оператор и после `else`.

Теперь разберемся, как писать условие. В языке C++ нет значка ≤ — вместо него необходимо писать <= (меньше или равно). Следовательно, на языке C++ наше условие выглядит так: `1 <= k <=9`. Однако в таком виде условие работать не будет. Чтобы убедиться в этом, напишем простую программу, текст которой приведен в листинге 1.3.

Листинг 1.3. Проверка заданных условий

```
#include <iostream.h>
int main()
{  int k;
   cout << "Введите целое значение от 1 до 9: ";
   cin >> k;    //  ввод числа
   if (1 <= k <= 9) cout << "Истинно" << endl;
   else cout << "Ложно" << endl;
   return 0;
}
```

Программа транслируется без ошибок и выполняется. Однако выясняется, что при любых значениях переменной `k` на экран всегда выводится слово "Истинно". Дело в том, что выражение `1 <= k <= 9` вычисляется последовательно слева направо. Поэтому сначала вычисляется выражение `1 <= k`, и вычисленный результат сравнивается с `9`.

Примечание

Здесь мы сталкиваемся с очень неприятной для программистов особенностью языка C++: неявным преобразованием типов. Результат вычисления выражения `1 <= k` преобразуется в целое число! Если выражение истинно, то результатом будет `1`; если же выражение ложно, то результат равен `0`. Теперь понятно, почему наше условие всегда истинно: и `0 < 9`, и `1 < 9`.

Вместо привычной математической записи мы должны в данном случае написать *логическое выражение*. Для этого сначала нужно разделить наше условие на два отдельных, а затем объединить их с помощью логической операции И. На языке C++ это выглядит так:

```
(1 <= k) && (k <= 9)
```

Немного переделаем условие в соответствии со стилем C++:

```
(0 < k) && (k < 10)
```

С точки зрения смысла ничего не изменилось, но такое выражение проще писать и легче читать. Скобки ясно выражают наши намерения. Таким образом, в листинге 1.4 представлен окончательный вариант нашей программы.

Листинг 1.4. Таблица умножения с проверкой множителя

```
#include <iostream.h>
int main()
{  int k;        //  объявление переменной
   cout << "Введите множитель от 1 до 9: ";
   cin >> k;     //  ввод числа
   if ((0 < k) && (k < 10))    //  внешние скобки обязательны
   {  cout << k << "* 1 =" << k * 1 << endl;    //  составной оператор
      cout << k << "* 2 =" << k * 2 << endl;
      cout << k << "* 3 =" << k * 3 << endl;
      cout << k << "* 4 =" << k * 4 << endl;
      cout << k << "* 5 =" << k * 5 << endl;
      cout << k << "* 6 =" << k * 6 << endl;
      cout << k << "* 7 =" << k * 7 << endl;
      cout << k << "* 8 =" << k * 8 << endl;
      cout << k << "* 9 =" << k * 9 << endl;
      cout << k << "* 10 =" << k * 10 << endl;
   }
   else cout << "Неправильное значение множителя!" << endl;
   return 0;
}
```

Напишем другую программу (ее текст приведен в листинге 1.5), в которой условный оператор используется более интенсивно. Пусть у нас есть кирпич с размерами сторон `a`, `b`, `c` и есть прямоугольное отверстие размером `x` на `y`. Необходимо проверить, пролезет ли кирпич в отверстие. Очевидно, что кирпич пройдет в отверстие, если любые две из его сторон меньше размеров отверстия. Упорядочим размеры отверстия `(x < y)`, и "выстроим" размеры сторон кирпича в возрастающем порядке: `a < b < c`. Тогда для ответа на вопрос, пройдет ли кирпич в отверстие, нужна одна проверка:

```
если (a < x) и (b < y) то ответ "ДА" иначе ответ "НЕТ"
```

Выстроить стороны кирпича в нужном порядке можно, обменяв значения переменных, как показано в следующей схеме:

```
если (a > b) то { t = a; a = b; b = t; }
если (b > c) то { t = b; b = c; c = t; }
если (a > b) то { t = a; a = b; b = t; }
```

Двух перестановок недостаточно. Пусть, например, `a = 10`, `b = 8`, `c = 6`. После первой проверки `a = 8`, `b = 10`, `c = 6`; после второй — `a = 8`, `b = 6`, `c = 10`. Аналогично можно поступить и с размерами отверстия, поэтому считаем, что `x < y`. Наши рассуждения почти "дословно" переводятся на язык C++.

Листинг 1.5. Программа "Кирпич и отверстие"

```
#include <iostream.h>
int main()
{  double a, b, c;    //  размеры кирпича
   cout << "Задайте размеры кирпича: ";
   cin >> a >> b >> c;
   double x, y;       //  размеры отверстия
   cout << "Задайте размеры отверстия: ";
   cin >> x >> y;
   if (x > y) { double t = x; x = y; y = t; }
   if (a > b) { double t = a; a = b; b = t; }
   if (b > c) { double t = b; b = c; c = t; }
   if (a > b) { double t = a; a = b; b = t; }
   if ((a < x) && (b < y)) cout << "Кирпич пройдет в отверстие!" << endl;
   else cout << "Кирпич не пройдет в отверстие!" << endl;
   return 0;
}
```

В программе, помимо использования условного оператора, мы видим еще несколько новых особенностей. Переменные объявляются по мере необходимости, а не в начале программы. В операторах ввода указаны сразу несколько переменных:

```
cin >> x >> y;
```

Такая запись просто короче, чем два оператора:

```
cin >> x;
cin >> y;
```

Как в том, так и в другом случае при вводе, значения можно задавать через пробел. Клавиша <Enter> нажимается после второго числа. Аналогично работает и оператор ввода размеров кирпича.

Самое интересное, что объявление переменной `t` четыре раза не вызывает "протестов" транслятора. Это объясняется тем, что данная переменная объявляется внутри составного оператора. Составной оператор с объявленными внутри переменными по многолетней традиции программирования называют *блоком*. В C++ принято, что в любом месте программы, где может стоять составной оператор, может стоять и блок. Поэтому мы в дальнейшем не будем делать различия между блоком и составным оператором и всегда будем называть конструкцию в фигурных скобках блоком. Переменные, объявленные в блоке, считаются *локальными переменными* в блоке и "живут от скобки до скобки" *(см. гл. 2)*.

Оператор-переключатель

В C++ есть еще один оператор выбора решения — оператор-переключатель. Напишем программу, представляющую упрощенную версию калькулятора. Программа вводит два числа и знак операции, которую требуется выполнить с этими числами. Знак операции — это один из символов +, –, *, /. Сначала напишем программу-калькулятор на C++, а затем разберем ее работу. Текст программы приведен в листинге 1.6.

Листинг 1.6. Калькулятор с оператором `switch`

```
#include <iostream.h >
int main()
{  cout << "Задайте два числа a и b: ";
   double a, b;
   cin >> a >> b;    //  ввод двух значений
   cout << "Задайте операцию [+ – * /]: ";
   char op;
   cin >> op;        //   ввод знака операции
   switch (op)       //  оператор-переключатель
   {  case '+': cout << a + b << endl; break;
      case '-': cout << a – b << endl; break;
      case '*': cout << a * b << endl; break;
      case '/': if (b != 0) cout << a / b << endl;
      else cout << "Делитель равен нулю!" << endl;
      break;
      default: cout << "Неверный знак операции!" << endl;
   }
   return 0;
}
```

Как обычно, в программе сначала вводятся необходимые данные, а потом символ операции, которую надо выполнить. Ввод символа по форме ничем не отличается от ввода чисел. После ввода написан оператор-переключатель. В общем виде оператор-переключатель имеет форму:

```
switch (переключающее_выражение)
{ набор альтернатив }
```

Переключающим выражением в нашей программе является переменная типа `char`, значение которой вводится с клавиатуры как символ. В общем виде одна *альтернатива* оператора имеет вид:

```
case константа: операторы;
```

Константа должна быть того же типа, что и переключающее выражение. В отличие от других операторов языка C++, в альтернативе можно задавать несколько операторов без фигурных скобок. Одна из альтернатив — особая, она начинается со слова `default`. Эту альтернативу можно не писать. Все слова — зарезервированные слова языка C++. Оператор-переключатель работает следующим образом:

- вычисляется переключающее выражение;
- значение выражения сравнивается поочередно с константами в альтернативах;
- первое совпадение приводит к выполнению операторов совпавшей альтернативы;
- выполняются все остальные операторы во всех остальных альтернативах, в том числе и в `default`.

После этого выполняется следующий за `switch` оператор. Однако на практике почти всегда нужно выполнять только *операторы совпавшей альтернативы*. Именно это требуется и в нашем случае. Для ограничения действия оператора-переключателя только одной альтернативой используется оператор `break`. Это тоже зарезервированное слово языка C++. Его действие заключается в немедленном выходе из блока — происходит неявный переход на оператор вне фигурных скобок оператора `switch`. Таким образом, оператор-переключатель в нашей программе работает так:

- значение переменной `op` по очереди сравнивается с константами в альтернативах;
- если произошло совпадение с одним из символов, выполняется соответствующий оператор вывода, в котором вычисляется нужное выражение;
- выполняется оператор `break`, что приводит к немедленному выходу из оператора-переключателя.

Если мы введем неправильный символ — знак операции, то совпадения не произойдет, и выполнится оператор в альтернативе `default`, который выведет сообщение об ошибке. В этой альтернативе нет необходимости задавать оператор `break`, т. к. после выполнения операторов этой альтернативы все равно происходит выход из оператора-переключателя.

Оператор-переключатель можно промоделировать с помощью вложенных условных операторов. Мы могли бы в программе-калькуляторе написать следующее:

```
if (op == '+') cout << a + b << endl;
else if (op == '-') cout << a – b << endl;
else if (op == '*') cout << a * b << endl;
else if (op == '/')
```

```
{ if (b != 0) cout << a/b << endl;
  else cout << "Делитель равен нулю!" << endl;
}
else cout << "Неверный знак операции!" << endl;
```

При такой записи необходимость в операторе `break` отпадает.

Следует отметить, что переключатель наиболее часто применяется для распознавания различных клавиш, вводимых с клавиатуры. Очень часто этот оператор используется при реализации *конечных автоматов* [3, 32, 35], где одна альтернатива оператора `switch` обозначает одно состояние автомата.

Повторение вычислений

Вернемся к программе "Таблица умножения с проверкой множителя", приведенной в листинге 1.4. Теперь наша программа делает именно то, что и требовалось. Сначала просит ввести множитель: если множитель правильный, то вычисляется и выводится на экран соответствующая таблица умножения; если множитель неправильный, то программа выводит сообщение об ошибке и заканчивает работу. Все правильно. Однако рассмотрим вывод таблицы умножения внимательнее. Операторы вывода отличаются один от другого всего в двух местах: в текстовой константе и в выражении умножения. Хотелось бы заменить десять очень похожих операторов одним-единственным, который отработает за все десять.

Это не пустой интерес программиста, которому лень написать десять почти одинаковых строк, это очень важная проблема. Представьте себе, что надо вычислять не десять вариантов, а 573. Или даже миллион! Если 573 еще можно написать вручную, то миллион мы просто не осилим: если в секунду писать по одному оператору, то в течение суток мы напишем 86 400 операторов. На всю программу у нас уйдет 2 недели! А представляете размер такой программы? Поэтому сокращение текста программы за счет однотипных вычислений — это очень важное действие.

Разумеется, это можно сделать, если повторить выполнение оператора вывода десять раз. И при каждом повторении (которые в программировании обычно называются *итерациями*) требуется изменить текстовую константу и второй множитель в выражении умножения. Для решения этих проблем мы заменим явную целую константу переменной. Эта переменная должна увеличиваться на `1` при каждом повторении. Назовем нашу переменную именем `i`. Значение переменной `i` представляет собой номер итерации. Обычно такие переменные называют *счетчиками*. В языке C++ принято (хотя это и не обязательно) начинать считать с нуля. С учетом этого соглашения мы можем написать схему так:

```
i = 0;
пока (условие истинно) увеличить i;
вывод одной строки таблицы умножения
```

Мы специально сдвинули две строки вправо, чтобы подчеркнуть тот факт, что эти строки должны повторяться, пока ... что? Разберемся с условием повторений. Последний раз повторение оператора вывода должно произойти при значении `i`, равном `9`. Как только `i` станет больше `9`, повторения должны прекратиться. Поэтому условие можно записать так: `i < 10`.

На языке C++ повторение выполнения можно организовать, если использовать *оператор цикла*. В C++ имеется три оператора цикла, но в данном случае мы используем *оператор с предусловием*, который записывается так:

```
while (условие) оператор;
```

Слово `while` является зарезервированным словом C++. Этот оператор цикла работает следующим образом: пока условие истинно, выполняется заданный оператор; как только условие стало ложным, заданный оператор пропускается и начинает работать следующий после цикла оператор. Условие записывается точно так же, как и в операторе `if`. И точно так же на месте заданного оператора можно прописать составной оператор в фигурных скобках. Составной оператор называется *телом цикла*. Наша схема на C++ будет выглядеть так:

```
i = 0;              //  начальное значение
while (i < 10)      //  условие продолжения
{  ++i;             //  увеличение переменной на 1
   cout << k << "*" << i << "=" << k * i << endl;
}
```

Мы преобразовали текстовую константу в операторе вывода. По сравнению с первоначальным вариантом от нее осталось только два неизменных символа: звездочка и равно, — остальное превратилось в переменные, имеющие разные значения в разный момент времени. Теперь можно написать полную программу с циклом, текст которой приведен в листинге 1.7.

Листинг 1.7. Таблица умножения с циклом `while`

```
#include <iostream.h>
int main()
{  int k;                        //  объявление переменной
   cout << "Введите множитель от 1 до 9: ";
   cin >> k;                     //  ввод числа
   if ((0 < k) && (k < 10))      //  внешние скобки обязательны
   {  int i = 0;                 //  начальное значение
```

```
      while (i < 10)            //  условие продолжения
     {  ++i;                    //  увеличение переменной на 1
        cout << k << "*" << i << "=" << k * i << endl;
     }
   }
   else cout << "Неправильное значение множителя!" << endl;
   return 0;
}
```

Как и в предыдущей версии исполнения программы, на экран выводится приглашение, и программа ждет ввода значения. Если задано недопустимое число, то программа выводит сообщение об ошибке и заканчивает работу. При вводе правильного значения выполняется составной оператор после условия оператора `if`. Сначала объявляется переменная `i`, которая сразу же получает значение `0`. В языке C++ переменные можно *инициализировать* сразу при объявлении.

Потом оператор цикла проверяет условие, которое в данном случае истинно (`0 < 10`). Следовательно, выполняется тело цикла. Сначала значение переменной `i` увеличивается на `1` и становится равным `1` — это делает оператор `++i`. После этого работает оператор вывода. Сначала выводится значение переменной `k`, потом (без пропусков) — символ `*`, значение `i` (в данном случае — единица), знак `=` (равно) и значение выражения `k * i`. Курсор экрана переводится на следующую строку. На этом месте тело цикла заканчивается, и программа возвращается к проверке условия. Условие (`1 < 10`) истинно, поэтому повторяется только что описанный процесс. И так продолжается до тех пор, пока при очередном увеличении значение переменной `i` не станет равным `10`. Тогда при проверке условие (`10 < 10`) окажется ложным (`10` не меньше `10`) и произойдет выход из цикла. И сразу же — выход из оператора `if`. Таким образом, будет выполняться оператор возврата — программа завершит работу.

Можно использовать и другой оператор цикла, в котором условие проверяется после выполнения тела цикла. Форма записи *цикла с постусловием* следующая:

```
do оператор;
while (условие);
```

Как обычно, на месте единственного оператора может стоять составной оператор в фигурных скобках. Важное отличие оператора цикла с постусловием от предыдущего варианта состоит в том, что в данном случае тело цикла обязательно выполнится хотя бы один раз, тогда как при вычислении предусловия может возникнуть такая ситуация, что тело цикла не сработает ни разу. Само условие вычисляется точно так же. Наша программа с циклом `do...while`, текст которой приведен в листинге 1.8.

Листинг 1.8. Таблица умножения с циклом `do...while`

```
#include <iostream.h>
int main()
{  int k;                        //  объявление переменной
   cout << "Введите множитель от 1 до 9: ";
   cin >> k;
   if ((0 < k) && (k < 10))      //  внешние скобки обязательны
   {  int i = 0;                 //  начальное значение
      do {  ++i;                 //  увеличение переменной на 1
            cout << k << "*" << i << "=" << k * i << endl;
         }
      while (i < 10);            //  условие продолжения
   }
   else cout << "Неправильное значение множителя!" << endl;
   return 0;
}
```

Как мы видим, программа фактически не изменилась, кроме непосредственно оператора цикла. Какой из операторов цикла использовать, зависит от задачи и вкусов программиста — язык C++ не накладывает никаких ограничений.

Рассматривая оба варианта программы, можно заметить, что организация цикла в обеих программах (см. листинг 1.7, 1.8) имеет некоторые сходные черты.

1. В том и в другом варианте имеется переменная-счетчик, которой один раз присваивается начальное значение.
2. Переменная увеличивается при каждой итерации, шаг увеличения равен `1`.
3. На каждом шаге цикла выполняется проверка условия.

Практика программирования показала, что такие циклы встречаются в программах очень часто. Поэтому разработчики C решили включить в язык третий вид оператора цикла, который так и называется: *цикл со счетчиком*. Его запись имеет вид:

```
for(начальное_выражение; условие; увеличение_счетчика) оператор;
```

`начальное_выражение` предназначено для инициализации переменной-счетчика цикла. `условие` записывается точно так же, как и в других операторах цикла. `увеличение_счетчика` — это именно то, что делает в наших циклах оператор `++i`. Естественно, вместо единственного оператора в теле цикла мы можем писать составной оператор в фигурных скобках.

Этот оператор цикла работает так:

- вычисляется `начальное_выражение`. Это происходит единственный раз при первом выполнении оператора `for`;
- проверяется `условие`. Если условие истинно, то выполняется заданный оператор в теле цикла. Если условие ложно, то происходит выход из цикла и выполняется следующий после цикла оператор;
- после выполнения заданного оператора вычисляется `увеличение_счетчика`;
- происходит переход к проверке условия.

Таким образом, при первой итерации выполняются первые два "пункта" в круглых скобках, а при последующих — второй и третий, только в обратном порядке. Текст программы с оператором `for` приведен в листинге 1.9.

Листинг 1.9. Таблица умножения с циклом `for`

```
#include <iostream.h>
int main()
{  int k;
   cout << "Введите множитель от 1 до 9: ";
   cin >> k;
   if ((0 < k) && (k < 10))
     for (int i = 0; i < 10; ++i)
     cout << k << "*" << i << "=" << k * i << endl;
   else cout << "Неправильное значение множителя!" << endl;
   return 0;
}
```

Программа стала значительно короче. У нас исчезли составные операторы с их скобками — программу стало легче читать. Отсюда вывод: если у вас в программе есть цикл со счетчиком — используйте оператор `for`.

Операторы *break* и *continue*

Операторы цикла можно использовать при проверке данных, задаваемых пользователем с клавиатуры. Идея такова: цикл должен выполняться до тех пор, пока пользователь не задаст правильное число. Для этого надо написать *бесконечный цикл*. Однако из цикла нужно выходить, если число введено правильно. Для этого обычно используется оператор `break`. Но в C++ есть и второй оператор, используемый исключительно внутри тела цикла. Это оператор `continue`. Посмотрите, как это можно использовать для нашей таблицы умножения:

```
int k = 0;
for(;;)                          //  бесконечный цикл
{  cout << "Задайте множитель от 1 до 9: ";
   cin >> k;
   if ((1 > k) || (k > 9))       //  неверное значение множителя
   continue;                     //  немедленный переход к проверке условия
   else break;                   //  немедленный выход из блока
}
```

В этом примере в условном операторе использован оператор `continue`, который немедленно передает управление на проверку условия, которое у нас пусто. Выражение увеличения счетчика тоже пусто, поэтому сразу выполняется оператор вывода приглашения. Если же условие оператора `if` не выполняется (число задано верно), то выполняется оператор `break`, и происходит немедленный выход из блока — тела цикла.

Бесконечный цикл можно задать и по-другому:

```
while(1) { ... }
```

Точно таким же образом можно использовать и цикл с постусловием.

Встроенные типы данных

Теперь настало время выяснить некоторые подробности. Начнем с чисел. Для того чтобы правильно использовать числа, программисту должны быть известны три вещи.

1. Как записывать константы.
2. Как объявлять переменные.
3. Какие операции можно выполнять с числами.

В большинстве языков программирования числа делятся на два больших класса: целые и дробные. Это обусловлено устройством компьютера. Более того, как целые, так и дробные числа подразделяются еще на несколько типов. Эта не очень приятная особенность аппаратуры естественно находит отражение в любом языке программирования, в т. ч. и в C++. Остановимся сначала на дробных числах.

Дробные числа

Дробные числа предназначены именно для вычислений. По исторически сложившимся причинам такие числа обычно называют числами "с плавающей точкой". Дробные константы в программе записываются в двух видах: обычном и научном. Обычный вид — он и есть обычный, только вместо

запятой, отделяющей дробную часть от целой, используется точка: `2.0`, `-123.4567`, `0.0`, — и т. п. Научный вид позволяет нам записывать или очень большие, или очень маленькие числа, например: `1.234e45`, `0.678e-23`. В этих константах само число называется *мантиссой*, а буква "e" (можно писать и большую букву "E") называется *экспонентой* и играет роль "10 в степени". Таким образом, первое число имеет значение 1.234×10^{45}, а второе — 0.623×10^{-23}. Понятно, что обычная форма записи здесь не подойдет — придется писать очень много нулей.

В C++ определено три "плавающих" типа: `float`, `double`, `long double` (все слова — зарезервированные слова языка C). Разрешается еще писать `long float`, но это просто более длинное обозначение `double`, поэтому мы не будем пользоваться такой записью. Дробные типы отличаются *диапазонами значений*. Диапазоны дробных чисел принято указывать от самых маленьких положительных чисел до самых больших. Отрицательные дробные числа имеют тот же диапазон, только со знаком минус. Если мы заглянем в справочник (Help) интегрированной среды Borland C++ 3.1, то увидим, что числа типа `float` занимают диапазон от 3.4×10^{-38} до $3.4\times10^{+38}$. Соответственно, числа типа `double` могут изменяться в диапазоне от 1.7×10^{-308} до $1.7\times10^{+308}$. И наконец, числа типа `long double` изменяются в диапазоне от 3.4×10^{-4932} до $1.1\times10^{+4932}$. Такие пределы также обусловлены конструктивными особенностями микропроцессоров фирмы Intel.

Примечание

В системе Borland C++ 3.1 точные пределы дробных чисел можно найти в файле float.h в каталоге include. В более современных системах (например, Visual C++ 6) пределы всех чисел прописаны в файле limits.

С первого взгляда трудно оценить, насколько велики или малы эти числа. Однако для сравнения можно привести расстояние от Земли до Солнца: в миллиметрах это расстояние приблизительно равно 1.45×10^{14}. Таким образом, даже самые "маленькие" числа с плавающей точкой намного превосходят это число.

Помимо диапазонов "плавающие" числа отличаются точностью. *Точность* — это количество знаков мантиссы. Числа типа `float` имеют 7—8 десятичных знаков, `double` — 15—16 знаков, а `long double` — 19—20. Обычно точности типа `double` хватает для большинства вычислений, но иногда требуется повышенная точность типа `long double`.

Каждый тип данных имеет свой размер. *Размер* — это количество единиц памяти, которое занимает объект программы. В микропроцессорах Intel единицей памяти является *байт*. Размер и определяет точность. Чтобы узнать размеры дробных чисел, надо выполнить программу, текст которой приведен в листинге 1.10.

Листинг 1.10. Определение размеров дробных чисел

```
#include <iostream.h>
int main()
{  cout << sizeof(float) << endl;
   cout << sizeof(long float) << endl;
   cout << sizeof(double) << endl;
   cout << sizeof(long double) << endl;
   return 0;
}
```

Слово `sizeof` является зарезервированным словом языка C++ и переводится именно как "размер" (чего-то). В языке C++ `sizeof` является операцией. Форма записи `sizeof(тип)` позволяет получить размер любого типа (как встроенного, так и определенного пользователем).

На экране мы увидим четыре числа `4`, `8`, `8` и `10` в столбик. Это означает, что числа типа `float` занимают в памяти 4 байта, `long float` и `double` — соответственно по 8 байт и `long double` — 10 байт.

Примечание

В системе Visual C++ 6 размер `long double` равен размеру `double(8)`.

Дробные константы тоже представляют числа одного из трех указанных типов. По умолчанию константы имеют тип `double`. Чтобы задать константы других двух типов, мы должны использовать *суффиксы*. Константа `3.141592653` — типа `double`, эта же константа с суффиксом F (или f) — `3.141592653F` — типа `float`, а константа с суффиксом L (или l) — `3.141592653L` — типа `long double`. Два суффикса одновременно писать нельзя. Размеры констант в памяти соответствуют их типам. Чтобы убедиться в этом, выполним программу, текст которой приведен в листинге 1.11.

Листинг 1.11. Определение размеров дробных констант

```
#include <iostream.h>
int main()
{  cout << sizeof 3.141592653F << endl;
   cout << sizeof 3.141592653 << endl;
   cout << sizeof 3.141592653L << endl;
   return 0;
}
```

Здесь использована вторая форма операции `sizeof`: `sizeof выражение` — выводится не значение выражения, а размер памяти. На экране появятся в

столбик те же числа 4, 8, 10 — это показывает, что константы имеют те же размеры.

С дробными числами в языке C++ можно выполнять обычные *арифметические* операции: сложение, вычитание, умножение, деление. Если x и y — дробные числа, то эти операции записываются так:

```
x + y    //  сложение
x – y    //  вычитание
x * y    //  умножение
x / y    //  деление
```

Во всех случаях результатом также является дробное число. К дробным числам можно применять операции *инкремента* ++ и *декремента* --, причем как в префиксной, так и в постфиксной форме. Операция инкремента увеличивает аргумент на 1, а операция декремента — уменьшает на ту же 1.

```
x++      //  инкремент – постфиксная форма
x--      //  декремент – постфиксная форма
++x      //  инкремент – префиксная форма
--x      //  декремент – префиксная форма
```

Различие этих операций проявляется в сочетании с операцией *присваивания*. Дробные числа можно сравнивать между собой:

```
x < y     //  x меньше y
x > y     //  x больше y
x <= y    //  x меньше или равно y
x >= y    //  x больше или равно y
x == y    //  x равно y
x != y    //  x не равно y
```

Примечание

Операция сравнения на равенство записывается как два знака равно. Начинающие программисты часто делают ошибку, записывая сравнение как один знак равно. Самое неприятное, что никаких сообщений об ошибке не появляется, но программа работает неверно. А дело все в том, что один знак равно обозначает совсем другую операцию — операцию присваивания, которую мы уже использовали и которую дальше рассмотрим подробнее.

Можно считать, что результатом операции сравнения является логическое значение "истина" или "ложь". Однако если мы попытаемся вывести логическое значение на экран

```
cout << (x < y) << endl;
```

то увидим либо 1, либо 0 в зависимости от того, выполняется ли это неравенство или нет. Такая несколько необычная интерпретация объясняется

историческими причинами: в C не было логического типа данных, поэтому вместо логических значений использовались целые константы `0` и `1`. Ноль означает "ложь", а единица — "истина". Именно поэтому мы написали `1` в условии бесконечного цикла `while`.

Примечание

В стандарте C++ определен логический тип `bool` и константы `true` и `false`. Однако по умолчанию при выводе на экран логических значений по-прежнему выдается `1` и `0`.

Переменные любых типов в C++ объявляются одинаково:

```
тип имя;
```

Можно через запятую задать несколько имен:

```
тип имя1, имя2, имя3;
```

Таким образом, дробные числа объявляются так:

```
float x, y, z;
double a, b, c;
long double v, w;
```

При объявлении переменных можно задавать *начальные значения*:

```
float t = 0.456, p = 1e-23;
```

Допускается в качестве начальных значений писать и *целые константы*:

```
long double x = 1, z = 0;
```

Вместо переменной можно объявить константу — в этом случае присваивать начальное значение уже обязательно:

```
const double d = 0.123456;
float const exponenta = 2.718281828;
```

Слово `const` является зарезервированным словом языка C++. Разрешается писать это слово как перед типом, так и после него. Изменить значение константы нельзя — при попытке сделать это программа даже не транслируется.

В C++ допускается и другая форма инициализации — в скобках. В скобках же можно задавать и значение константы:

```
float t(0.456), p(1e-23);
long double x(1), z(0);
const double d(0.123456);
```

Применение дробных чисел покажем на примерах вычисления школьных формул длины окружности, площади круга и объема шара. Для большей

универсальности будем вводить радиус с клавиатуры. Еще нам потребуется константа π (пи). Мы, конечно, можем задать ее явным образом, но мы не будем этого делать, поскольку обычно интегрированная среда содержит эту константу в стандартной библиотеке. В среде фирмы Borland C++ 3.1 константа π включена в математическую библиотеку и имеет имя `M_PI`.

Примечание

Удивительно, но в системе Visual C++ 6 математические константы не определены! Поэтому их надо задать самостоятельно тем или иным способом.

Подключение математической библиотеки выполняется точно так же, как и библиотеки ввода/вывода — директивой препроцессора `#include`.

Текст программы для математических вычислений приведен в листинге 1.12.

Листинг 1.12. Математические вычисления

```
#include <math.h>    //  математическая библиотека
#include <iostream.h>
int main()
{  double r = 0.0;
   label: cout << "Задайте радиус: ";
   cin >> r;         //  ввод числа
   if (r < 0)
   {  cout << "Радиус не может быть меньше нуля!" << endl;
      goto label;
   }
   double L = 2*M_PI*r;           //  длина окружности = 2*π*r
   cout << "Длина окружности = " << L << endl;
   double S = M_PI*r*r;           //  площадь круга = π*r^2
   cout << "Площадь круга = " << S << endl;
   double V = 4/3*M_PI*r*r*r;     //  объем шара = 4/3*π*r^3
   cout << "Объем шара = " << V << endl;
   return 0;
}
```

Радиус не может быть отрицательным, поэтому выполняется необходимая проверка вводимого значения. Тут мы использовали новый оператор *перехода* `goto`. Работа этого оператора состоит в безусловном переходе на указанную метку. *Метка* — это идентификатор, который не требуется специально объявлять, достаточно написать его в нужном месте и поставить после него двоеточие. Таким образом, программа, проверяя введенное значение, не

позволит пользователю продолжить работу, пока он не задаст правильное значение. Как видим, оператор `goto` с меткой эквивалентен бесконечному циклу, который мы рассматривали ранее.

Целые числа

Существует три типа целых чисел в языке C++:

- короткие `short`;
- "нормальные" `int`;
- длинные `long`.

Предполагается, что три типа целых чисел отличаются диапазонами и размерами. Но если мы выполним программу определения размеров для целых чисел в интегрированной среде Borland C++ 3.1 (ее текст приведен в листинге 1.13), то окажется, что размеры составляют 2, 2 и 4 байта соответственно.

Листинг 1.13. Определение размеров целых чисел

```
#include <iostream.h>
int main()
{  cout << sizeof(short) << endl;
   cout << sizeof(int)<< endl;
   cout << sizeof(long) << endl;
   return 0;
}
```

Если мы выполним ту же программу в среде Visual C++ 6 или Borland C++ Builder 6, то увидим числа `2`, `4`, `4`. Таким образом, единственное, что можно гарантировать, это выполнение неравенства:

```
Sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
```

Автор языка Б. Страуструп в книге [37] сообщает, что размеры и диапазоны целых чисел зависят от реализации — стандарт почти ничего не регламентирует в этом случае. Гарантируется только то, что для типа `short` используется, по меньшей мере, 2 байта, а для `long` — 4 байта.

Кроме того, целые числа обычно бывают *знаковыми* (`signed`) и *беззнаковыми* (`unsigned`). Заметим, что дробные числа бывают только знаковыми. Беззнаковые целые — это натуральные числа, наименьшее значение которых равно нулю. Еще раз выполним программу определения размера с учетом "знаковости", текст которой приведен в листинге 1.14.

Листинг 1.14. Определение размеров целых чисел с учетом "знаковости"

```
#include <iostream.h>
int main()
{  cout << sizeof(short) << endl;
   cout << sizeof(int) << endl;
   cout << sizeof(long) << endl;
   cout << sizeof(signed short) << endl;
   cout << sizeof(signed int) << endl;
   cout << sizeof(signed long) << endl;
   cout << sizeof(unsigned short) << endl;
   cout << sizeof(unsigned int) << endl;
   cout << sizeof(unsigned long) << endl;
   return 0;
}
```

Как выясняется, "знаковость" и "беззнаковость" никак не влияет на размер соответствующего типа, однако диапазоны чисел зависят от размера и знаковости.

Примечание

В системе Borland C++ 3.1 размеры и диапазоны целых чисел указаны в файле limit.h, который находится в каталоге include.

Целые константы можно задавать в трех системах счисления: *шестнадцатеричной, восьмеричной* и *десятичной*. Десятичные константы начинаются с любой цифры, кроме нуля. Если константа начинается с нуля, то константа считается восьмеричной. В записи такой константы могут присутствовать только восьмеричные цифры от `0` до `7`. Шестнадцатеричные константы начинаются с символа `0x` или `0X`. Как известно, в качестве цифр в шестнадцатеричной системе используются десятичные цифры от `0` до `9` и первые шесть букв латинского алфавита `A`, `B`, `C`, `D`, `E`, `F` или `a`, `b`, `c`, `d`, `e`, `f`.

Примечание

Шестнадцатеричные и восьмеричные константы, так же, как и битовые операции, редко бывают нужны в прикладном программировании. Как правило, в шестнадцатеричной системе представляются адреса и значения различных *флагов*, которые на самом деле являются отдельным битом.

Независимо от системы счисления, у нас есть возможность задавать "нормальные" знаковые константы типа `int`, беззнаковые константы типа `int`, длинные константы со знаком, длинные константы без знака. Типы констант, так же как и для дробных чисел, задаются суффиксами U (или u) и L (или l): `123` — обычная десятичная константа со знаком, `123U` — беззнаковая десятичная константа типа `int`, `123L` — длинная знаковая константа, `123LU`

(или `123ul`) — длинная константа без знака. Короткие константы не могут быть заданы никакими способами.

Наличие большого количества зарезервированных слов, относящихся к целым типам, позволяет объявлять переменные целого типа разнообразными способами:

```
int k;
short x, y, z;                //  int можно не писать
long a = 0, b = 1;            //  int можно не писать
unsigned int w = 2, v(10);    //  полная спецификация типа
signed long pp, rr(-5);       //  signed можно не указывать
unsigned long int t = 10;     //  полная спецификация типа
unsigned f;                   //  беззнаковый int
```

На самом деле целый тип — только `int`, остальные слова называются *модификаторами*. Но было бы слишком "жестоко" заставлять программиста каждый раз писать *полную спецификацию* типа, поэтому разрешается указывать сокращенные варианты типов: `short`, `long`, `signed` и `unsigned`. Если типы `short`, `long`, `int` не указаны, то по умолчанию считается тип `int`. Если отсутствуют типы `signed` и `unsigned`, по умолчанию подразумевается тип `signed`.

Так же как и для дробных чисел, аналогичным образом можно объявлять и целые константы различных типов, которые потом нельзя изменить:

```
const unsigned int BakersDozen = 13;          //  чертова дюжина
unsigned int const AnotherBakersDozen(13);    //  чертова дюжина
```

Целые константы можно задавать и другим способом — с помощью конструкции `enum`, например:

```
enum { one = 1, two = 2, ten = 10 };
```

Такая запись эквивалентна следующему объявлению:

```
const int one = 1, two = 2, ten = 10;
```

Если мы хотим объявить несколько констант с последовательными целыми значениями, то можно не присваивать значение константам, кроме первой, например:

```
enum { one = 1, two, three, four, five, six, seven, eight, nine, ten };
```

Каждая следующая константа имеет значение на 1 больше, чем предыдущая. Если наш ряд констант начинается с нуля, то можно вообще не присваивать значения, т. к. первая константа по умолчанию получает значение 0, например:

```
enum { zero, one, two, three, four, five, six, seven, eight, nine, ten };
```

Можно объявлять и отрицательные константы:

```
enum { bit = -1, zero };
```

Принцип остается тот же — следующая константа имеет значение на `1` больше по сравнению с предыдущей. Таким образом, константа `zero` имеет значение ноль.

Конструкция `enum` может быть именованной. Мы с помощью нее можем объявить новый (целочисленный) тип данных, например:

```
enum Boolean { false, true };
```

Такое описание вводит новый тип, переменные которого мы можем объявлять так:

```
Boolean a;
Boolean b(true);
Boolean d = false;
```

В логических выражениях такие переменные будут работать правильно, однако это не совсем тот тип данных, который бы нам хотелось иметь. Переменные типа `Boolean` являются целочисленными, а не логическими, поэтому программист не застрахован от ошибок. Например, без всяких сообщений об ошибках транслятор "скушает" следующие операторы:

```
a = b + d;
int f = b + false;
```

и им подобные. Очевидно, это не совсем то, что программист ожидает от логического типа данных.

Мы имеем возможность и другим способом задать константы "истина" и "ложь", используя оператор `typedef`.

```
typedef int Boolean;
const Boolean false = 0;
const Boolean true = 1;
```

Это тоже позволяет использовать имена `true` и `false` в сравнениях и логических выражениях.

Примечание

Стандарт C++ уже включает логический тип `bool`. Поэтому `true` и `false` стали зарезервированными словами, и системы Borland C++ Builder 6, Visual C++ 6 не пропустят такие объявления. Но в Borland C++ 3.1 это вполне допустимо.

Оператор `typedef`, с которым мы еще неоднократно столкнемся, в своем простейшем виде представляет собой следующую конструкцию:

```
typedef старое_имя_типа новое_имя_типа;
```

Если мы привыкли работать на Turbo Pascal, то с помощью этого оператора можем объявить привычные нам типы `word` или `real`, например:

```
typedef unsigned int word;
typedef double real;
```

Именно таким образом в Windows добавлено огромное количество "типов" вроде DWORD, BOOL или BYTE.

По стандарту разрешено на месте `новое_имя_типа` через запятую задавать несколько имен — все они являются синонимами старого.

С целыми числами допускаются те же операции сравнения и те же арифметические операции, что и с дробными. Операция деления выполняется нацело — дробная часть отбрасывается. Например, все следующие выражения дадут результат, равный `8`: `32/4`, `33/4`, `34/4`, `35/4`.

В языке C++ есть одна операция, которая применяется только к целым аргументам: *получение остатка от деления* (или деление по модулю), которая обозначается символом `%` (процент). Особенности этой операции показывает простая программа, текст которой приведен в листинге 1.15.

Листинг 1.15. Деление по модулю

```
#include <iostream.h>
int main()
{  cout << 5 % 2 << endl;
   cout << -5 % 2 << endl;
   cout << 5 % -2 << endl;
   cout << -5 % -2 << endl;
   return 0;
}
```

Во всех системах результаты получаются следующие: `1`, `-1`, `1`, `-1`. Как мы видим, знак остатка зависит только от знака делимого и не зависит от знака делителя. Эту операцию можно использовать для определения четности-нечетности числа. Если в целой переменной `k` хранится некоторое целое число, то определить, является ли оно четное или нечетное, можно следующим образом:

```
if (k % 2 == 0) cout << "Число четное!" << endl;
```

Часто эта операция используется при обработке дат — дней, месяцев и лет. Напишем программу, определяющую, является ли год високосным. Как известно, високосными годами являются все года, которые делятся на 4 без остатка (остаток равен нулю), за исключением тех, которые делятся на 100. Если год делится на 100, то он является високосным только в том случае,

если делится на 400. Таким образом, високосными годами являются 1600, 2000, 2004, 1996. А 1900 или 2003 годы не являются високосными. Очевидно, что переменная, в которой должно храниться значение-год, должна быть целая беззнаковая. Учитывая все эти соображения, можно написать нашу программу так (листинг 1.16).

Листинг 1.16. Високосный год

```
#include <iostream.h>
int main()
{  unsigned int year = 0;       //  инициализация
   begin: cout << "Задайте год: ";
   cin >> year;
   if (year == 0) goto end;     //  заканчиваем работу
   if ((year % 100) != 0)
     if ((year % 4) == 0) cout << "Год високосный!" << endl;
     else cout << "Год не високосный!" << endl;
   else
     if ((year % 400) == 0) cout <<"Год високосный!" << endl;
     else cout << "Год не високосный!" << endl;
   goto begin;                  //  начнем сначала
   end:
   return 0;
}
```

В этой программе мы видим несколько новшеств, которые раньше не применялись. Во-первых, программа будет продолжать работу до тех пор, пока мы не введем ноль. Это достигнуто использованием двух операторов `goto` и двух меток: `begin` в начале и `end` в конце. Во-вторых, мы использовали вложенные условные операторы. Схема вложенных операторов `if` выглядит так:

```
if (условие) if (условие) оператор; else оператор;
```

Поскольку при такой записи неясно, к какому `if` относится `else`, в языке C++ принято соглашение, что такая запись эквивалентна следующей:

```
if (условие) { if (условие) оператор; else оператор; }
```

При такой записи становится совершенно очевидно, что `else` связывается с ближайшим оператором `if`.

Мы можем упростить нашу программу, если вместо использования нескольких условных операторов объединим все условия в одном:

```
если ((год не делится на 100) и (год делится на 4))
       или (год делится на 400),
то год високосный;
```

В скобках мы задали отдельные составные части условия високосности, которые были распределены по отдельным условным операторам. Заодно можно ликвидировать лишние операторы вывода на экран:

```
if (((year % 100) != 0) && ((year % 4) == 0) || ((year % 400) == 0))
cout << "Год високосный!" << endl;
else cout << "Год не високосный!" << endl;
```

Логическую операцию И (`&&`) мы уже ранее использовали, а в этом операторе в условии применяется логическая операция ИЛИ (`||`).

Для проверки программы надо задать четыре варианта значения-года: 2 високосных и два не високосных. Последовательно введем значения `2000`, `1996`, `1700`, `1997`. В первых двух случаях программа должна сообщить, что год високосный, в остальных — не високосный. Для завершения работы программы введем ноль.

Символы и строки

Как мы уже видели, строковые константы в языке C++ пишутся в двойных кавычках. Внутри кавычек, как уже было сказано, можно использовать любые символы, которые можно набрать на клавиатуре, в т. ч. и русские буквы.

Примечание

При выводе на экран строка выводится именно так, как она записана внутри кавычек только в системе Borland C++ 3.1. В более поздних системах, работающих под Windows, необходимо учитывать разницу в кодировках символов.

Однако встроенного строкового типа в C++ нет. А вот символьный тип появился еще в C и является встроенным в C++. Константы-символы записываются в одиночных кавычках (знак апострофа): `'a'`, `'+'`, `'п'`. Символы внутри апострофов можно указывать любые. Однако символьную константу-апостроф так написать нельзя — система Borland C++ выводит сразу несколько сообщений об ошибках. Тем не менее, C++ предоставляет конструкцию для представления любых символов. Это делается с помощью обратной косой черты `\` (backslash). Апостроф в программе на C++ изображается как `'\"'`, кавычки — `'\"'`, сама обратная косая черта — `'\\'`. Тот же механизм используется для представления не изображаемых символов. Большинство из этих символов управляют курсором на экране дисплея; например, `'\b'` означает клавишу <Backspace>, а `'\r'` — перевод курсора в начало строки. Звуковой сигнал можно подать константой `'\a'`, выполнив оператор вывода:

```
cout << '\a' << endl;
```

Вместо манипулятора `endl` можно использовать символьную константу перевода строки `'\n'`, которая выполняет точно такие же действия, как и манипулятор. Более того, после косой черты `\` мы можем задавать целую константу в восьмеричном или шестнадцатеричном виде, например `'\15' = '\xD'` — код клавиши <Enter>. Такие константы занимают в памяти 1 байт, в отличие от обычных целых, которые занимают `sizeof(int)` байт.

Символ табуляции `'\t'` часто используется для выравнивания вывода на экране, что демонстрирует программа вычисления объединенной таблицы умножения для всех чисел сразу, текст которой приведен в листинге 1.17.

Листинг 1.17. Полная таблица умножения

```
#include <iostream.h>
int main()
{  cout << "Таблица умножения" << endl
   << "1 \ t2 \ t3 \ t4 \ t5 \ t6 \ t7 \ t8 \ t9" << endl
   << "---------------------------------"
   << "---------------------------------" << endl;
   for (int i = 1; i < 10; i++)
   {  cout << i << "| ";
      for (int j = 1; j < 10; j++) cout << j * i << '\t';
      cout << endl;
   }
   return 0;
}
```

Вывод этой программы выглядит следующим образом:

```
Таблица умножения
   1    2    3    4    5    6    7    8    9
-------------------------------------------------------------------
1| 1    2    3    4    5    6    7    8    9
2| 2    4    6    8    10   12   14   16   18
3| 3    6    9    12   15   18   21   24   27
4| 4    8    12   16   20   24   28   32   36
5| 5    10   15   20   25   30   35   40   45
6| 6    12   18   24   30   36   42   48   54
7| 7    14   21   28   35   42   49   56   63
8| 8    16   24   32   40   48   56   64   72
9| 9    18   27   36   45   54   63   72   81
```

Символьные константы имеют тип `char`. Переменные типа `char`, как и константы, занимают в памяти 1 байт. Мы можем объявлять переменные типа `char` либо знаковыми (`signed`), либо беззнаковыми (`unsigned`). Если мы не

указываем "знаковость-беззнаковость", то по умолчанию устанавливается тот атрибут, который задан в интегрированной среде (обычно это знаковые символы).

Значения, которые содержат символьные переменные, являются целыми числами. Эти "маленькие целые числа" называются кодами символов. Пока код находится в памяти, он воспринимается как обычное целое число, с которым позволяется выполнять любые операции, допустимые с целыми числами. Но как только мы попытаемся вывести это число на экран, аппаратура вместо числа выводит соответствующий символ. Чтобы убедиться в этом, выполним простую программу, текст которой приведен в листинге 1.18.

Листинг 1.18. Вывод символа по коду

```
#include <iostream.h>
int main()
{  unsigned char nn = 48;    //  символьная переменная
   cout << nn << ' ' << int(nn) << endl;
   return 0;
}
```

В этой программе объявлена беззнаковая символьная переменная, которой присвоена целая константа. Однако при выводе символьной переменной на экран выводится не числовое значение, а цифра `'0'`. И это не зависит от "знаковости-беззнаковости".

Чтобы вывести число, мы должны прописать *явное преобразование типа* `int (nn)`. Тем самым мы сообщаем транслятору, что желаем выводить данную переменную не в виде по умолчанию (как символ), а именно как целое число. Допускается задавать преобразование типа и в стиле C:

```
cout << nn << ' ' << (int)nn << endl;
```

на экран, как и в первом случае, будет выведено целое число.

Каждый не изображаемый символ также имеет свой код. Мы легко убедимся в этом, включив в нашу программу следующие операторы:

```
cout << int('\t') << endl;    //  табуляция
cout << int('\n') << endl;    //  новая строка
cout << int('\a') << endl;    //  звуковой сигнал
cout << int('\b') << endl;    //  backspace
```

На экран в столбик будут выведены следующие числа: `9`, `10`, `8`, `7`. Принцип кодирования символов целыми числами позволяет нам, объявив символьную переменную, по мере необходимости менять в ней значение кода символа. Выполним простую программу, текст которой приведен в листинге 1.19.

Листинг 1.19. Вывод целого как символа

```
#include <iostream.h>
int main()
{  for (char i = 48; i < 59; ++i) cout << i << ' ';
   return 0;
}
```

Программа выводит символы `0 1 2 3 4 5 6 7 8 9`. Таким образом, у нас получилось, что коды символов-цифр представляют собой целые числа от 48 до 57.

Поскольку коды — целые числа, мы можем оперировать символьными данными как целыми. При этом нет нужды прописывать преобразование типа. Пусть, например, у нас в переменной `ch` содержится код любой цифры. Тогда выражение

```
ch - '0'
```

позволяет получить значение цифры.

Если вас смущает возможность арифметических операций с символами, объявите с помощью `typedef` новые символьные типы:

```
typedef signed char small;
typedef unsigned char byte;
```

Теперь для арифметических операций можно использовать типы `small` и `byte`, а для символов, как обычно, `char`.

Множество целых чисел, используемых для представления символов в памяти компьютера, обычно называют *кодировкой символов*. Работая в системе Borland C++ 3.1, мы пользуемся американским стандартным кодом ASCII. Хотя с английским алфавитом проблем, как правило, не возникает, тем не менее приходится констатировать, что единообразия здесь не наблюдается. Даже на одном и том же IBM PC используется несколько кодировок — все зависит от операционной системы. Поэтому при переносе программы на другую платформу, в другую операционную систему обязательно приходится учитывать особенности кодирования символов. А если нам придется писать программу с поддержкой нескольких языков, то необходимо учитывать особенности национальных кодировок.

Операции присваивания и выражения

Создатели языка C в свое время очень новаторски подошли к вычислительным возможностям языка. В язык включено много необычных операций. В выражениях можно смешивать разные операции. Присваивание — это тоже операция. В простейшем виде присваивание выглядит так:

```
имя = выражение
```

Символ = и является обозначением операции присваивания. Выражение справа от знака равенства вычисляется и значение заносится в переменную, имя которой указано слева. При этом могут выполняться *неявные преобразования типа*. Мы сознательно не поставили в конце точку с запятой, подчеркнув тем самым, что присваивание — это операция, а вся эта запись сама является выражением. Мы можем превратить это выражение в оператор, поставив в конце точку с запятой. Ситуация в данном случае совершенно аналогичная использованию инкремента: `++x` — это выражение, а `++x;` — это оператор. Такой подход является несколько необычным по сравнению с другими языками программирования, где присваивание — оператор по определению.

Первым следствием такого подхода является возможность осуществлять *многократное присваивание*:

```
имя_1 = имя_2 = имя_3 = ... = имя_n = выражение
```

Такая запись тоже является выражением и выполняется справа налево: выражение вычисляется, значение заносится в переменную `имя_n`, потом в предыдущую переменную и т. д. до переменной `имя_1`. Поскольку присваивание — операция, ее можно смешивать с другими операциями в выражениях, например:

```
a = 5 + (b = 6);
```

В этом операторе переменная `b` получает значение `6`, а переменная `a` становится равной `11` (`5 + 6`). Другой необычный пример:

```
a = (b > 5);
```

Переменная `a` получит значение `0` или `1` в зависимости от истинности или ложности условия. На самом деле это не условие, а логическое выражение.

Теперь разберемся с разницей префиксной и постфиксной форм инкремента (и декремента). Пусть в переменной `x` находится значение `5`. Рассмотрим два оператора присваивания:

```
d = ++x;
```

и

```
d = x++;
```

В первом случае переменная `x` увеличится на `1` и новое значение `6` запишется в переменную `d`. Второй оператор присваивания работает совершенно по-другому: сначала число `5` присваивается переменной `d`, а потом значение переменной `x` увеличивается на `1`. Таким образом, второй оператор на самом деле эквивалентен двум операторам:

```
d = x;
++x;
```

Нужно отметить, что "лишние" скобки в данном случае не помогают: оператор

```
d = (x++);
```

работает точно так же — сначала число 5 попадает в d, потом x увеличивается на 1. Таким образом, несмотря на использование одинакового обозначения, постфиксная и префиксная формы инкремента (декремента) работают совершенно по-разному.

Изменение переменной не на единицу, а на некоторую другую константу (два, три) тоже встречается достаточно часто. В программах обычно приходится выполнять такое присваивание:

```
переменная = переменная + число;
```

В этой записи и слева, и справа от символа = (равно) указана одна и та же переменная. Например, в программе может быть следующая запись:

```
i = i + 2;
```

Выполняется этот оператор так: к значению переменной i (например, 3) прибавляется константа 2 и полученное значение (5) заносится в ту же переменную. Создатели языка C решили включить в язык специальные *арифметические операции с присвоением*. Обозначаются подобные операции двойным символом #=, где решетка обозначает один из знаков арифметических операций: +, -, *, /, %. Общая форма записи таких операций следующая:

```
переменная #= выражение
```

Такая форма означает:

```
переменная = переменная # (выражение)
```

Сначала вычисляется выражение, потом выражение и переменная участвуют в операции #, а затем результат заносится в переменную. Таким образом, приведенный выше способ увеличения переменной на 2 на языке C++ позволит записать выражение короче:

```
i += 2;
```

На мой взгляд, такая запись еще и более понятна. Как и "простая" операция присваивания, такие операции тоже являются многократными.

Преобразование типов

Мы уже сталкивались с преобразованием типов, как явным, так и неявным. Неявные преобразования выполняются при вычислениях выражений и присваивании. Вкратце правила преобразования по умолчанию можно описать такой цепочкой:

```
[[char, short] -> int] -> unsigned int -> long ->
unsigned long -> float -> double -> long double
```

Как видно, менее "объемный" тип преобразуется к более "объемному". В скобках записано обязательное преобразование типов `char` и `short` перед выполнением операции — эти типы всегда преобразуются в `int`, а затем выполняется операция.

По умолчанию может выполняться и обратное преобразование, например:

```
float -> int
```

однако подобного рода преобразования связаны с потерей информации (дробная часть просто отбрасывается), поэтому хороший компилятор обязательно об этом предупреждает.

Явные преобразования мы уже неоднократно выполняли. Операция преобразования может записываться в двух коротких формах:

```
(тип)выражение
тип(выражение)
```

С точки зрения результата эти две формы *абсолютно эквивалентны*. Однако Б. Страуструп отмечает, что "явное преобразование типа ... используется неоправданно часто и является основным источником ошибок" [37, с. 172]. Поэтому в стандарт C++ включены новые *не взаимозаменяемые* операторы преобразования типа:

```
static_cast<тип>(выражение)
reinterpret_cast<тип>(выражение)
```

Первый оператор осуществляет преобразование типов с проверкой на этапе компиляции. Например, все преобразования в приведенной выше цепочке можно выполнять явно при помощи оператора `static_cast`. Второй оператор выполняет преобразование типов без проверки при компиляции. На практике такое преобразование необходимо тогда, когда точно известно, что объекты разных типов занимают в памяти одинаковое количество байт и фактического преобразования делать как раз не нужно. Б. Страуструп предупреждает [37, с. 172], что выражение преобразования с оператором `reinterpret_cast` "практически никогда не переносимо". Примеры использования этих операторов мы увидим в дальнейшем.

Есть еще один оператор преобразования `dynamic_cast`, который выполняет преобразование на этапе выполнения — его мы изучим в *части II*.

Другие операции

Одной из необычных операций является *условная операция* `?`, единственная из операций, имеющая три аргумента:

```
выражение_условие? выражение_1: выражение_2
```

Выполняется она так: вычисляется выражение_условие, если результат не равен нулю, то вычисляется выражение_1 и выдается в качестве значения операции; если результат вычисления условия нулевой, то вычисляется выражение_2. Эта операция позволяет делать условные вычисления значительно короче. Например, выбор максимума из двух чисел можно записать так:

```
max = (a > b) ? a : b;
```

Мы уже пользовались логическими операциями && (логическое И) и || (логическое ИЛИ). Есть еще и третья логическая операция — логическое отрицание, обозначаемое символом ! (восклицательный знак). Эта операция, единственная из всех логических операций, является *унарной*. Все эти операции используются в логических выражениях. Результатом логического выражения будет 0 (false — ложь) или 1 (true — истина). Эти операции могут применяться к любым типам данных. Логическое отрицание подчиняется следующим правилам:

```
!0 = 1    !false = true
!1 = 0    !true = false
```

Логическое ИЛИ выполняется по следующим правилам:

```
0 || 0 = 0    false || false = false
0 || 1 = 1    false || true = true
1 || 0 = 1    true || false = true
1 || 1 = 1    true || true = true
```

Логическое И выполняется по следующим правилам:

```
0 && 0 = 0    false && false = false
0 && 1 = 0    false && true = false
1 && 0 = 0    true && false = false
1 && 1 = 1    true && true = true
```

Операции логического ИЛИ и логического И являются сокращенными: второй аргумент не вычисляется, если уже по первому аргументу ясен результат. Например, выражение

```
(1 || выражение)
```

всегда будет равно 1, поэтому после вычисления первого аргумента нет нужды вычислять второй. Аналогично выражение

```
(0 && выражение)
```

всегда равно 0, поэтому второй аргумент также можно не вычислять. Более формально с использованием условной операции это можно выразить так:

- выраж_1 && выраж_2 эквивалентно (выраж_1?(выраж_2?1:0):0);
- выраж_1 || выраж_2 эквивалентно (выраж_1?1:(выраж_2?1:0));
- !выражение эквивалентно (выражение?0:1).

Создатели языка C пошли еще дальше (а язык C++ унаследовал эти особенности): условие в условном операторе и в операторах цикла (и в условной операции) на самом деле являются просто выражением, результат которого не обязательно должен быть истинным или ложным. Результат может быть любым числом, в т. ч. дробным или даже отрицательным. Если результат не равен нулю, то условие считается истинным, в противном случае — ложным. Таким образом, определение условного оператора в C++ на самом деле следующее:

```
if (выражение) оператор;
```

Этот оператор эквивалентен:

```
if (выражение != 0) оператор;
```

Оператор

```
if (выражение == 0) оператор;
```

можно записать короче

```
if (!выражение) оператор;
```

Именно поэтому не выдается никаких сообщений, если программист написал вместо операции == (равно) операцию = (присвоить). Например, следующий оператор абсолютно правилен с точки зрения синтаксиса:

```
if (a = 0) cout << "Переменная a равна 0" << endl;
else cout << "Переменная a не равна 0" << endl;
```

однако работает вопреки элементарной логике — всегда выдается сообщение о том, что переменная `a` не равна `0`. А все дело в пропущенном знаке равенства: выражение в условии стало не сравнением, а присваиванием, результат которого равен `0`. Следовательно, по правилам работы условного оператора всегда выполняется альтернатива `else`. Чтобы избежать подобной ловушки, некоторые авторы советуют писать константу слева, а переменную — справа:

```
if (0 == a) cout << "Переменная a равна 0" << endl;
else cout << "Переменная a не равна 0" << endl;
```

Тогда условие с одним знаком равно будет ошибочным, и компилятор сразу это обнаружит.

Аналогичное определение принято и для операторов цикла. Например, цикл с предусловием определен в C++ следующим образом:

```
while (выражение) оператор;
```

Выражение вычисляется и проверяется при каждой итерации: пока выражение не равно нулю, итерации цикла будут продолжаться. Подобную же картину мы наблюдаем в операторе `for`, который по определению имеет такой вид:

```
for (выражение_1; выражение_2; выражение_3) оператор;
```

выражение_1 называется инициализирующим и вычисляется только один раз при первой итерации цикла; начиная со второй итерации, вместо первого вычисляется выражение_3. Вычисление выражение_2 выполняется на каждом шаге — именно результат вычисления этого выражения определяет, будет ли продолжаться выполнение цикла. Аналогично другим выражениям-условиям ненулевой результат означает повторение выполнения тела цикла.

Продемонстрируем применение этих операций, а также необычность подхода к вычислениям в языке, написав программу вычисления среднего арифметического чисел, вводимых с клавиатуры. Нам потребуется переменная для суммы и переменная для количества введенных чисел. Программа должна работать для любого количества чисел, и мы заранее не знаем, сколько их будет. Числа могут быть как положительные, так и отрицательные (листинг 1.20).

Листинг 1.20. Цикл while с операцией "запятая"

```
#include <iostream.h>
int main()
{  unsigned int counter = 0;
   double summa = 0.0, number;
   while (cout << "Введите число: ", cin >> number)    //  правильное
                                                       //  условие
   summa += number, ++counter;                         //  один оператор
   cout << "Среднее=" << summa / counter << endl;
   return 0;
}
```

В программе написан совершенно необычный оператор цикла. В условии прописаны сразу и вывод приглашения, и ввод числа через запятую. Дело в том, что и вывод приглашения, и ввод значений на самом деле являются выражениями. В операторы они превращаются только тогда, когда мы ставим в конце точку с запятой. И запятая в языке C++ тоже является операцией, причем с двумя аргументами! Результатом этой операции является результат второго (правого) выражения. Таким образом, везде, где можно написать одно выражение, через запятую можно написать несколько выражений. Окончательным результирующим значением ряда выражений является значение последнего из них. Поэтому оператор в нашей программе будет выполняться до тех пор, пока мы не нажмем специальную комбинацию клавиш <Ctrl>+<Z> (и потом клавишу <Enter>). Эта комбинация означает "конец ввода", в результате чего значение последнего выражения — ввода станет равным 0, и оператор цикла прекратит свою работу.

Тело цикла содержит единственный оператор, в котором, однако, вычисляются два выражения. В первом выражении накапливается сумма введенных чисел, а во втором — подсчитывается их количество.

Теперь напишем ту же программу с использованием оператора `for` (ее текст приведен в листинге 1.21.

Листинг 1.21. Цикл `for` с операцией "запятая"

```
#include <iostream.h>
int main()
{  unsigned int counter = 0;
   for (double summa = 0.0, number;                    //  это не оператор
   (cout << "Введите число: ", cin >> number);         //  это не оператор
   summa += number, ++counter);                        //  пустое тело
   cout << "Среднее=" << summa / counter << endl;
   return 0;
}
```

Мы разбили оператор `for` на три строки только из соображений читабельности. Количество операторов в программе еще уменьшилось — объявления двух переменных мы поместили на место первого выражения оператора `for`. На месте первого выражения допускается объявлять (и инициализировать) несколько переменных одного типа. Два выражения, обеспечивающие ввод числа, помещены на место второго выражения, а на месте третьего оказалось тело оператора `while` из предыдущего варианта программы. Тело цикла осталось пустым.

После вычисления `выражение_1` сначала вычисляется `выражение_2`. В нашем случае последовательно сработает сначала вывод на экран приглашения, затем программа будет ждать ввода числа от пользователя. Если пользователь не будет нажимать комбинацию клавиш <Ctrl>+<Z>, то в переменную `number` запишется число, и оператор `for` будет выполнять операторы тела — а там пусто! Поэтому сразу произойдет переход к вычислению `выражение_3` — именно то, что и требовалось. После вычисления всех выражений, стоящих на месте `выражение_3`, происходит переход к вычислению `выражение_2`, и программа опять требует ввода числа.

С использованием операции "запятая" операцию постфиксного инкремента

```
a = x++;
```

можно записать так:

```
a = (t = x, x += 1, t);
```

где `t` должна иметь такой же тип, как и `x`.

Операций в языке C++ много (см. табл. 1.1), некоторые из них (как и запятая) являются не совсем обычными. Например, вызов функции *(см. гл. 2)* — это выражение, в котором операцией (бинарной) являются круглые скоб-

ки `()`. Индексирование массива `[]` *(см. гл. 3)* тоже является операцией. Каждая операция имеет свой *приоритет*, от которого и зависит *порядок вычисления* выражения. В табл. 1.1 операции перечислены в порядке убывания приоритета: в первой строке указаны операции с наивысшим приоритетом, в последней — с наименьшим.

Таблица 1.1. *Операции C++*

Операции	Порядок выполнения
`() [] -> :: .`	Слева направо
`! ~ + - ++ -- & * sizeof new delete`	Справа налево
`.* ->*`	Слева направо
`* / %`	Слева направо
`+ -`	Слева направо
`<< >>`	Слева направо
`< <= > >=`	Слева направо
`== !=`	Слева направо
`&`	Слева направо
`^`	Слева направо
`|`	Слева направо
`&&`	Слева направо
`||`	Слева направо
`?:`	Слева направо
`= *= /= %= += -= &= ^= |= <<= >>=`	Справа налево
`,`	Слева направо

Печальная действительность

До сих пор все наши программы прекрасно работали. Однако в программировании полно "подводных камней", о которых начинающий программист не подозревает. Одной из первых проблем, с которыми приходится столкнуться, является *переполнение*. Допустим, в нашей программе есть переменная `a` типа `float`, которая содержит величину, близкую к предельной, например, 10^{37}. Что произойдет, если мы попытаемся выполнить такой оператор:

```
cout << a * 1000 << endl;
```

Пока мы запускаем программу в интегрированной среде Borland C++ 3.1, операция не выполняется. Оператор выводит на экран значение `1e+37`. Но стоит нам запустить программу независимо от среды, мы получаем сообщение:

```
Floating point error: Overflow (Ошибка плавающей точки: Переполнение)
Abnormal program termination (Ненормальное завершение программы)
```

Очевидно, результат просто не может поместиться в переменную типа `float`. Такие ситуации называются переполнением. Понятно, что аналогичная картина может возникнуть и при использовании других плавающих типов. Интегрированная среда просто отслеживает такую ситуацию и не дает программе совершить аварию.

Рассмотрим аналогичную ситуацию с целыми числами. Заглянув в помощь системы Borland C++ 3.1, обнаруживаем, что целые числа типа `int` изменяются от −32 768 до +32 767. Посмотрим, что получится, если мы будем работать на границах диапазона:

```
cout << +32767 + 1 << endl;
cout << -32768 - 1 << endl;
```

Первый оператор выводит на экран отрицательное число `-32 768`, а второй — положительное число `32 767`. Таким образом, мы наблюдаем, что при добавлении `1` к правой (положительной) границе диапазона получается отрицательное число — левая граница диапазона. При вычитании `1` из левой (отрицательной) границы диапазона получаем, соответственно, правую границу диапазона. От размера чисел эта ситуация не зависит — то же самое происходит и с числами типа `long` и `short`.

Аналогичную картину наблюдаем и для беззнаковых чисел `unsigned int`, которые изменяются в диапазоне от 0 до 65 535. Оператор

```
cout << 65535u + 1 << endl;
```

выдает на экран ноль. Аналогично, оператор

```
cout << 0u - 1 << endl;
```

выводит число `65 535`. Самое неприятное, что не выводится никаких сообщений или предупреждений ни при трансляции, ни при выполнении программы.

С типом `char` — аналогичные проблемы, которые усугубляются принятой по умолчанию "знаковостью-беззнаковостью". Диапазон "маленьких чисел" изменяется от −128 до 127. Однако ни системы фирмы Borland, ни Visual C++ 6 не "протестуют", если в программе встречается, например, такой оператор:

```
char ch = 255;
```

Вывод `ch` как целого числа показывает на экране `-1`. Справедливости ради надо сказать, что Visual C++ 6 выдает предупреждение:

```
warning C4305: 'initializing' : truncation from 'const int' to 'char'
'инициализация': усечение 'const int' до 'char'
```

Иногда недоумение вызывают совершенно безобидные выражения. Например, переменная `d` в следующем операторе

```
double d = 15/2;
```

будет равна `7`, а не `7,5`. А все дело в том, что выражение `15/2` — это деление целых чисел и выполняется с отсечением. И только после выполнения операции результат (целое число) преобразуется в дробное типа `double`. Чтобы таких ошибок не было, необходимо в выражениях задавать дробные константы:

```
double d = 15.0/2;
```

или

```
double d = 15/2.0;
```

Операция "запятая" тоже может спровоцировать ошибку, например:

```
double d;
d = 3,141592653;
```

Переменная `d` равна `3`, а не "пи", как задумывал программист — вместо точки написана запятая. Интересно, что при инициализации

```
double d = 3,141592653;
```

возникает ошибка трансляции.

"Страшный зверь" по имени *undefined behavior*

Как указано в стандарте языка C++ [50], порядок вычисления подвыражений внутри выражений не определен. Не стоит полагаться на то, что вычисления выражений выполняются слева направо. Это не противоречит указанному в табл. 1.1 приоритету операций. Пусть у нас в программе написано выражение:

```
a = b + c + d;
```

где `b`, `c` и `d` сами являются выражениями. Операции сложения действительно будут выполняться слева направо, т. е. сначала к `b` прибавится `c`, а потом к результату прибавится `d`. Однако порядок вычислений выражений `b`, `c` и `d` — не определен. Компилятор может сначала вычислить `c`, затем `d` и только потом `b`. Собственно, ну и пусть, скажете вы. Тем не менее не все так просто. Рассмотрим несколько искусственный, но простой пример:

```
int i = 1, k = 2, j = 3;
i = (i = j - k) + (i = j + k);
cout << i << endl;
```

Ничего "криминального" не наблюдается, системы фирмы Borland (Borland C++ 3.1, Borland C++ 5, Borland C++ Builder 6) во всех режимах трансляции выдают один и тот же результат 6. Поменяв местами слагаемые

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

тоже во всех вариантах получаем 6. Вроде бы все логично: независимо от порядка вычисления выражение j - k должно иметь значение 1, а j + k равно 5. Поэтому результат получается равным 6. Если бы я писал транслятор, я именно так и рассуждал бы при реализации арифметических выражений. Однако разработчики Visual C++ 6 думали иначе: то же выражение

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

в этой среде имеет значение 10. А если поменять слагаемые местами, тогда результат получается равным 2. Такие ситуации в стандарте называются *undefined behaviour* — неопределенное поведение. Основной отличительной чертой является неоднократное присваивание одной и той же переменной в пределах одного выражения. Например, следующее выражение является как раз таким:

```
(a += b) += a;
```

Какое значение переменной a будет добавляться к результату в скобках: то, которое было до изменения a, или уже измененное? Неопределенное поведение может проявляться не только в операциях с присваиванием, но и при использовании инкремента или декремента. Например, оператор

```
i = ++i + 1;
```

является примером из стандарта, демонстрирующим неопределенное поведение. Вообще-то говоря, невинные, на первый взгляд, выражения

```
++++i; (++i)++;
```

тоже отличаются свойством неоднократного присваивания, однако во всех системах, где я их испытывал, получался один и тот же результат — увеличение i на 2. С другой стороны, оператор

```
i = 7, i++, i++;
```

в стандарте указан как пример *defined behaviour* — определенное поведение. Переменная i в данном случае равна 9. Таким образом, операция "запятая" может использоваться для "придания определенности" вычислениям. Другими операциями, гарантирующими определенный порядок вычисления выражений, являются логические операции И (||) и ИЛИ (&&).

Таким образом, многие вычисления необходимо явно проверять в той среде, с которой вы в данный момент работаете. Мы еще неоднократно будем возвращаться к проблеме неопределенного поведения в других главах книги.

ГЛАВА 2

Функции

Подпрограммы появились, несомненно, во "второй день творения". Это было так давно, что современные программисты уже практически забыли сам термин "подпрограмма". Тем не менее это настолько фундаментальное понятие, что без него невозможно представить ни один серьезный язык программирования. Мы уже посчитали, что написать действительно большую программу достаточно трудно. Как бы то ни было, такие программы пишут, и они успешно работают. Добиться этого можно только одним способом — разделить программу на части и поручить работу нескольким программистам. Таким образом, нам в языке программирования требуются средства, позволяющие выделять часть программы в отдельную конструкцию. Исторически первой такой конструкцией были подпрограммы.

Уже в первом широко известном языке Fortran были определены и реализованы два типа подпрограмм: процедуры и функции. В настоящее время такие языки программирования, как Pascal, делают различие между функциями и процедурами и предоставляют программисту два вида подпрограмм. В C++ же процедуры отсутствуют, и функции играют обе роли.

Второй причиной появления подпрограмм стала необходимость сохранять и накапливать уже сделанную работу. Компьютеры работали настолько медленно, что потребовалось придумать целую технологию, позволяющую не транслировать программу лишний раз. Так появились *библиотеки стандартных подпрограмм*.

В стандарте языка C++ определена богатейшая стандартная библиотека, которая, помимо традиционного набора стандартных подпрограмм, включает большой набор разнообразных структур данных с соответствующими операциями обработки. Ряд стандартных подпрограмм реализован настолько универсально, что их стали называть *стандартными алгоритмами*.

Важнейшими вопросами при изучении подпрограмм являются следующие: как определить подпрограмму, как вызвать ее для выполнения, какие типы

параметров допустимы, каким образом они передаются в подпрограмму и как возвращаются результаты, допускается ли рекурсия. Однако помимо этих, традиционных для любого языка программирования вопросов, C++ включает еще ряд интересных особенностей, отсутствующих в других языках, например, функции с переменным числом параметров, перегрузка функций, механизм шаблонов и ряд других.

Библиотечные функции

Язык C++ предоставляет программисту колоссальное количество стандартных функций. Все они в совокупности составляют стандартную библиотеку C++, которая состоит из большого количества отдельных тематических библиотек. Программисты, говоря о стандартной библиотеке, часто имеют в виду стандартную библиотеку шаблонов (Standart Template Library, *STL*). Но на самом деле STL — это только часть стандартной библиотеки, хотя и достаточно большая.

Для того чтобы использовать любую функцию, необходимо точно знать, в какой библиотеке она находится. В мире C++ имеются следующие "источники" функций:

- стандартные библиотеки, оставшиеся в "наследство" от C;
- стандартные библиотеки C++, составляющие STL;
- дополнительные библиотеки интегрированной среды, например, полезная, но не входящая в стандарт библиотека conio.h;
- объектно-ориентированные библиотеки интегрированной среды (MFC, ATL, WTL, OWL, VCL), которые используются при программировании для Windows;
- API — множество функций, предоставляемых операционной системой;
- нестандартные библиотеки, например, boost.

Чтобы использовать функцию, мало знать, в какой библиотеке она находится. Мы должны знать, как называется функция, какие параметры необходимы для ее правильной работы и какой результат возвращает функция. Например, очевидно, что результат функции, вычисляющей синус угла, будет дробным числом от −1 до +1. Однако она может иметь аргумент в градусах, а может и в радианах. Если же мы зададим аргумент неправильного типа, то и ответ будет неверный — функция-то не может разобраться, что за число мы ей "подсунули". В языке C++ для указания всей этой информации используется объявление функции, которое выглядит так:

```
тип имя(список параметров);
```

`имя` — это имя функции, оно является обычным идентификатором C++. Список параметров определяет количество, порядок, типы, способ передачи

и, возможно, начальные значения параметров. Параметры в объявлении обычно называются *формальными*. Если параметров нет, то используется зарезервированное слово `void`.

`тип` определяет тип возвращаемого значения. Вызов функции в C++ — это выражение, поэтому по типу возвращаемого значения можно определить, в каких выражениях разрешается использовать вызов этой функции. Вызов имеет вид:

```
имя(список аргументов)
```

Аргументы, задаваемые при вызове функции, называются *фактическими параметрами*. Список фактических параметров, как правило, должен в точности соответствовать списку формальных параметров, хотя в C++ возможны исключения. Функция, возвращающая значение некоторого типа, обычно должна вызываться как один из элементов выражения того же типа, например:

```
double y = sin(x) + 2 * cos(x) + 1.53;
```

Если функция не возвращает значения, то вместо типа нужно задавать зарезервированное слово `void`. Функции, не возвращающие значений, аналогичны процедурам в других языках программирования, и вызов таких функций должен быть задан отдельным оператором, например:

```
randomize();
```

Вообще-то так можно вызывать любую функцию, но для функций, возвращающих значение, это практически не имеет смысла — значение просто теряется.

По традиции объявление функции в C++ также называют *прототипом*. Прототипы всех стандартных функций, которые разрешается использовать в той или иной интегрированной среде, можно найти во включаемых текстовых файлах, расположенных в каталоге include интегрированной среды.

Математические функции

Мы начнем изучение функций C++ с использования стандартных математических функций, которые в интегрированных средах фирмы Borland собраны в библиотеке math.h. Мы уже использовали эту библиотеку, когда нам требовалась константа π (пи) для вычисления длины окружности и площади круга (см. листинг 1.7). Пусть нам требуется по трем сторонам `a`, `b` и `c` треугольника вычислить три его высоты `ha`, `hb`, `hc`. Как известно, высота, опущенная на сторону `a`, вычисляется по формуле

$$h_a = \frac{2}{a}\sqrt{p(p-a)(p-b)(p-c)}$$

где `p = (a+b+c)/2` — это полупериметр треугольника. Квадратный корень представляет собой формулу Герона для вычисления площади треугольника. Заглянем в помощь интегрированной среды: в библиотеке math.h обнаруживаем два прототипа:

```
double sqrt(double x);
long double sqrtl(long double x);
```

Совершенно очевидно, что обе функции, несмотря на разные имена, вычисляют квадратный корень из своего аргумента. Различается только тип аргумента и тип результата — вторая функция считает результат с повышенной точностью. Повышенная точность нам не нужна, поэтому будем использовать первый вариант.

Нам необходимы переменные для сторон треугольника, высот и полупериметра. Стороны треугольника мы задаем с клавиатуры, остальное — вычисляем. Мы не будем проверять задаваемые значения на допустимость (хотя в реальных программах обязательно надо это делать) — понятно, что стороны треугольника не могут быть отрицательными или нулевыми. Требуется также выполнение неравенства треугольника. Напишем программу, текст которой приведен в листинге 2.1.

Листинг 2.1. Вычисление высот треугольника

```
#include <iostream.h>
#include <math.h>
int main()
{  double a, b, c;               //  стороны
   double ha, hb, hc;            //  высоты
   double p = 0, S = 0;          //  полупериметр и площадь
   cout << "Задайте стороны треугольника: ";
   cin >> a >> b >> c;
   p = (a+b+c)/2;                          //  вычисляем полупериметр
   S = sqrt(p * (p-a) * (p-b) * (p-c));    //  вызов функции — вычисляем
                                           //  площадь
   ha = 2/a*S;
   cout << "Ha =" << ha << endl;
   hb = 2/b*S;
   cout << "Hb =" << hb << endl;
   hc = 2/c*S;
   cout << "Hc =" << hc << endl;
   return 0;
}
```

Сначала мы вычисляем полупериметр, затем площадь, а потом — высоты. Правильность работы нашей программы легче всего проверить на специ-

альных видах треугольников: прямоугольном, равностороннем, равнобедренном. Для прямоугольного треугольника со сторонами 3, 4, 5 программа совершенно правильно выдает две высоты, равные катетам 3 и 4, а высота, опущенная на гипотенузу, равна 2.4. Для равностороннего треугольника со сторонами 1, 1, 1 высоты получаются одинаковыми: 0.866025. Для равнобедренного треугольника со сторонами 3, 3, 2 получаются две высоты размером 1.885618, а третья высота равна 2.828427.

В библиотеке math.h собрано довольно много функций: все стандартные тригонометрические, обратные тригонометрические и гиперболические функции; несколько функций вычисления логарифмов и возведения в степень; ряд функций позволяет получать ближайшие целые значения от дробных. В этой же библиотеке собраны и основные математические константы.

В состав любой интегрированной среды C++ входит стандартная библиотека stdlib.h. В этой библиотеке тоже есть ряд полезных функций, в частности, для получения случайных чисел. Случайные числа полезны при отладке программы, поэтому мы научимся использовать эти числа на примере простой игры, описанной в [44], которую мы немного модифицируем. Программа "задумывает" число в диапазоне от 1 до 1000, а мы будем его угадывать. Выбирается число с использованием функции для генерации случайных чисел. Сначала надо вызвать функцию инициализации, а затем функцию генерации случайного числа. В интегрированной среде Borland C++ 3.1 инициализация случайной последовательности может быть выполнена функцией randomize, имеющей следующий прототип:

```
void randomize(void);
```

Как мы уже знаем, это означает, что функция не имеет параметров и не возвращает значения. Для генерации случайного числа воспользуемся функцией random, которая имеет такой прототип:

```
int random(int num);
```

Примечание

Эти функции не являются стандартными ни в C, ни в C++, однако входят в состав системы Borland Pascal, и фирма Borland включила их в состав своих "сишных" систем. В составе системы Visual C++ 6 этих функций нет.

Функция генерирует случайное число в диапазоне [0, num). Такая запись означает, что num не входит в указанный интервал: последним числом интервала будет num−1. Таким образом, чтобы генерировать случайное число в диапазоне от 1 до 1000, мы должна написать такой вызов:

```
int number = random(1000) + 1;
```

Вообще, чтобы генерировать случайные числа в диапазоне от A до B, надо вызывать функцию random следующим образом:

```
int number = random(B − A) + A;
```

Напишем программу, текст которой приведен в листинге 2.2.

Листинг 2.2. Угадай число

```
#include <iostream.h>
#include <stdlib.h>
int main()
{  cout << "Задумано число от 1 до 1000." << endl;
   cout << "Попробуйте угадать!" << endl;
   cout << "Вам дается не более 10 попыток" << endl;
   const int n = 10;          //  максимальное число попыток
   randomize();               //  инициализация датчика случайных чисел
   unsigned int number = random(1000) + 1;    //  случайное число
   unsigned int getnum;       //  получаемое от пользователя число
   int i = 0;
   while (i < n)
   {  cout << "Какое число задумано?" << endl;
      cin >> getnum;
      ++i;    //  считаем количество попыток
      if (number < getnum) cout << "Задуманное число меньше!\n";
      if (number > getnum) cout << "Задуманное число больше!\n";
      if (number == getnum)
      {  cout << "Вы угадали!!!\n";
         cout << "Сделано " << i <<" попыток.\n";
         return 0;     //  завершение с успехом
      }
   }
   cout << "Вы исчерпали все попытки! Жаль." << endl;
   return -1;          //  завершение с неудачей
}
```

Угадывать число мы будем, применяя двоичный поиск. Задаем середину диапазона `500`, программа отвечает: задуманное число больше или меньше заданного. Допустим, задуманное число меньше `500`. Тогда следующее задаваемое число `250` — середина диапазона 0—500. Если программа отвечает, что задуманное число больше `500`, то следующее задаваемое число `750`, являющееся серединой диапазона 500—1000. Этот процесс очень быстро приведет нас к "задуманному" числу.

Как написать функцию

Как всякий "уважающий себя" язык программирования, C++ предоставляет программисту массу возможностей писать свои собственные функции. Каждая функция, как и в других языках программирования, состоит из заголов-

ка и тела. *Заголовок* — это прототип, только в конце точка с запятой не ставится. В заголовке задается имя функции, список параметров, тип возвращаемого значения. *Тело функции* — это блок. Тело функции определяет *алгоритм* работы. Напишем простую функцию, вычисляющую максимум из трех целых чисел.

```
int max3(int a, int b, int c)
{  int t = (a > b) ? a: b;
   return (t > c) ? t: c;
}
```

Заголовок функции понятен. В теле функции определена локальная переменная `t`, которая при объявлении проинициализирована выражением выбора максимума `a` и `b`. Далее написан уже известный нам оператор возврата, в котором в качестве возвращаемого значения написано выражение окончательного выбора максимума.

Ту же функцию можно написать значительно короче (ее текст приведен в листинге 2.3), если использовать стандартную функцию вычисления максимума из двух чисел, прототип которой можно найти в stdlib.h.

Листинг 2.3. Функция выбора максимума из трех целых чисел

```
int max3(int a, int b, int c)
{ return max (max (a, b), c); }
```

В теле функции написан только один оператор возврата, в котором все и выполняется. Вызов функции `max3` выполняется совершенно аналогично вызовам стандартных функций, например:

```
int r = max3(x, 5, z);
```

При вызове этой функции необходимо прописывать в программе оператор

```
#include <stdlib.h>
```

т. к. функция `max` объявлена в этой библиотеке. Еще одним простым примером является функция вычисления длины окружности по заданному радиусу. Определение функции может быть таким:

```
double LengthCircle(double r) { return 2*M_PI*r; }
```

Совершенно аналогично выглядит и определение функции, вычисляющей площадь круга по радиусу:

```
double SquareCircle(double r) { return M_PI*r*r; }
```

Вызов этих функций ничем не отличается от вызова любой стандартной математической функции.

В качестве примера "процедуры" напишем нашу программу вывода таблицы умножения в виде функции. Очевидно, что функция не вычисляет никакого полезного значения для последующего использования, поэтому можно написать слово `void` в качестве типа возвращаемого значения. Параметром является множитель таблицы умножения. Следуя принципу *инкапсуляции*, мы должны поручить проверку параметров самой функции. В таком случае функция должна возвращать некоторое значение, которое покажет, получила ли функция правильный параметр или нет. Пусть возвращаемое значение `0` сигнализирует о правильном параметре, а значение `-1` — о том, что параметр неверен. Перенесем в функцию оператор проверки параметра. Текст функции приведен в листинге 2.4.

Листинг 2.4. Функция "Таблица умножения"

```
int TableMult(int k)
{  if ((0 < k) && (k < 10))
  {  for ( int i = 1; i < 10; ++i)
     cout << k << "*" << i << "=" << k * i <<endl;
     return 0;    //  нормальный возврат
  }
else return -1;   //  параметр неверен
}
```

Главная функция в этом случае выглядит таким образом:

```
int main()
{  int k, cc;
   cout << "Введите множитель от 1 до 9: ";
   cin >> k;
   if (cc = TableMult(k))
   cout << "Неправильное значение множителя! " << endl;
   return cc;
}
```

Как видите, функция `main` значительно упростилась. Вызов функции выполняется прямо в условии: если множитель задан правильный, то функция выполняет свою работу и возвращает ноль. Поэтому вывод сообщения об ошибке не выполняется. Если же множитель задан неверно, то функция возвращает `-1`, и на экране появляется сообщение об ошибке.

Определение функции

Теперь поговорим о формальностях. Каждая функция в программе на C++ должна быть один раз определена и может быть объявлена несколько раз по мере необходимости. По правилам языка C++ функция должна быть опре-

делена (или объявлена) до того, как использована (вызвана). В приведенных примерах мы видели, что определение функции всегда написано до первого вызова.

Определение функции, как уже было сказано, состоит из двух частей: заголовка и тела. В общем случае заголовок функции включает следующие части:

- спецификаторы класса памяти: `extern` или `static`. По умолчанию — `extern`;
- тип возврата, возможно `void`. По умолчанию — `int`;
- модификаторы;
- имя функции;
- список объявления параметров;
- спецификацию исключений `throw(список типов исключений)`.

Спецификация класса памяти влияет на область видимости имени функции. Спецификатор `extern` означает, что объявленная функция доступна из любого места программы. Работу спецификатора `static` мы рассмотрим при изучении организации многомодульных программ. Модификаторы, вообще говоря, полностью зависят от реализации и нам практически не понадобятся. Спецификация исключений определена в стандарте и старыми компиляторами не поддерживается. Мы изучим исключения в *части II*. Остальные элементы заголовка мы уже рассматривали.

Тело функции — это блок, т. е. последовательность описаний переменных и операторов, заключенных в фигурные скобки. Даже если функция не выполняет никаких действий, тело функции должно присутствовать в определении. В этом случае тело функции будет состоять просто из скобок `{ }`:

```
void f(void) { }
```

Такая функция может быть необходима при отладке в качестве "заглушки".

Примечание

Определения функций не должны быть вложенными. В отличие от определений, прототипы — объявления функции — могут быть вложенными в другие функции. Главное, чтобы прототип был прописан до вызова соответствующей функции.

Например:

```
float Max(float x, float y)              //  определена до вызова
{ return (x < y ? y : x); }
int main(void)
{  float Cube(float a);    //  прототип до определения вложен в главную
   int cc=  15, k = 2;
```

```
   k = Cube(k);
   cc = 5 + Max(k, cc) / 3 + Cube(k);     //  вызовы функций
   cout << cc;
}
float Cube(float x)                       //  определение функции
{ return (x * x * x); }
```

Как видите, имена параметров в прототипе и в определении функции могут не совпадать. Более того, в прототипе можно вообще не указывать имен параметров, достаточно указать типы, однако имена повышают читабельность программы.

Список параметров и вызов функции

Рассмотрим теперь технические подробности передачи параметров в функцию. В заголовке определения функции формальные параметры перечисляются через запятую. Наиболее часто спецификация отдельного параметра имеет синтаксис, аналогичный синтаксису обычных объявлений переменных:

```
тип имя
```

Тип аргумента может быть любым, который только возможен в C++: либо одним из встроенных простых типов, либо типом, определенным пользователем тем или иным способом. Имя — обычный идентификатор, если параметров несколько, то все имена должны быть различны. Определение параметров все-таки отличается от определения переменных: если при объявлении переменных можно для нескольких имен указать один тип, то при объявлении параметров для каждого имени указывается свой собственный тип.

Стандарт языка C++ разрешает указывать спецификацию формального параметра без имени. Естественно, такой параметр невозможно использовать в теле функции.

Обычно считается, что в C++ существует три способа передачи параметров.

1. Передача *по значению*.
2. Передача *по указателю*.
3. Передача *по ссылке*.

На самом деле, первые два способа — это наследие языка C, и в C++ их можно не использовать, поскольку механизм передачи по ссылке позволяет осуществить и тот и другой способ более элегантно и эффективно. Новичок может задать естественный вопрос: нельзя ли обойтись каким-нибудь одним способом? Нельзя — мы увидим, что в разных ситуациях требуются различные способы передачи параметров.

Передача параметров по значению

Передача по значению — самый распространенный и надежный метод передачи, работающий так же, как и во многих других языках программирования. При такой передаче параметров предполагается, что каждый объявленный в заголовке параметр является локальной переменной в теле функции *(см. разд. "Область видимости и "время жизни" переменных" данной главы)*.

Это имеет такие следствия при вызове:

- передача по значению эквивалентна операции присваивания `имя_параметра = выражение`. Выражение вычисляется, если необходимо (и возможно), приводится к типу параметра и присваивается параметру. Если на месте выражения стоит константа или переменная, совпадающая по типу с параметром, то значение просто копируется в него;
- копирование значения фактического параметра в локальную переменную требует времени, поэтому способ передачи параметров по значению обычно применяется для данных простых типов, время копирования которых мало;
- если фактический параметр является переменной, то любые изменения параметра внутри функции не влияют на переменную вне функции.

Во все стандартные математические функции: `sin`, `cos`, `exp` и др., — параметры передаются по значению. Все ранее написанные нами функции использовали именно такой способ передачи параметров. В качестве примера рассмотрим определенную ранее простую функцию `Cube`, вычисляющую куб своего аргумента. Аргумент в ней как раз передается по значению.

```
float Cube(float x)             //  определение функции
{ return (x * x * x); }
float a = 5.5, b = 1.23, c = Cube(2);     // фактический
                                          //  параметр — константа
float d = Cube(b);              // фактический параметр — переменная
float e = Cube(a/b + 2.51);     //  фактический параметр — выражение
float f = Cube(Cube(c));        //  параметр — выражение — вызов функции
```

Все примеры понятны, за исключением последней строки. С математической точки зрения это вызов означает `f(f(x))`. Поскольку `c` равно `8`, то получается следующий результат: `8` в кубе равно `512`, которое при возведении в куб даст значение `134217728`.

Наглядно продемонстрировать, что передача параметра по значению эквивалентна операции присваивания, позволяет нам постфиксная операция инкремента.

Пусть в программе объявлена некоторая переменная b, имеющая значение 1. Тогда операторы

```
cout << Cube(b++) << endl;
cout << b << endl;
```

выведут на экран

```
1
2
```

Несмотря на то, что параметр передается по значению, сначала в функцию передается значение единица, а вычисление выражения b++ выполняется после передачи значения параметра в функцию — это в точности соответствует выполнению операции присваивания:

```
x = b++;
```

Как правило, список фактических параметров должен соответствовать списку формальных параметров по количеству и типам. В C++ возможны исключения из этого правила при определении параметров *по умолчанию* и списка параметров *переменной длины*, которые будут рассмотрены далее. Однако язык C++ и помимо этих возможностей допускает совершенно "экзотические" (с точки зрения синтаксиса) способы написания фактических параметров (их текст приведен в листинге 2.5)[1].

Листинг 2.5. Несколько выражений на месте одного параметра

```
double sqr(double x)                  //  определен один параметр
{ return x * x; }
int main(void)
{  double k = 0;
   double g = 1, f=2;
   g = sqr((f = sqr(f), k += f));     //  на месте параметра — два
                                      //  выражения
   cout << "\n\n f =" << f;
   cout << "\n g =" << g;
   cout << "\n k =" << k;
  return 0;
}
```

Вызов g = sqr((f = sqr(f), k += f)) не вызывает протестов компилятора, хотя визуально вместо одного параметра наблюдается два. Здесь используется уже известное нам свойство языка C++: символ "запятая" является операцией. Как мы помним, это позволяет в любом месте, где допускается

[1] Понятно, что для работы надо добавить #include.

выражение, через запятую написать любое количество выражений. Значением списка выражений является значение последнего (крайнего правого) выражения. "Лишние" скобки здесь совсем не лишние, поскольку без них будет ошибка синтаксиса: два параметра вместо объявленного одного.

Такой вызов работает следующим образом. До вызова функции `sqr` значение переменной `f` равно `2`, а переменная `k` имеет значение, равное нулю. Вычисляется выражение `f = sqr(f)` и переменная `f` получает значение `4`, которое используется в следующем выражении для изменения значения переменной `k`. Значение выражения `k += f`, передаваемое во "внешний" вызов функции `sqr`, равно `4`. После этого переменной `g` присваивается значение `16 = 4 × 4`. Таким образом, данная программа выведет на экран следующее:

```
f = 4
g = 16
k = 4
```

Параметры по умолчанию

В C++ разрешается задавать значение параметров по умолчанию. Для простых типов при передаче параметров по значению синтаксис присвоения значения по умолчанию выглядит следующим образом:

```
тип имя = выражение
```

Как мы видим, объявление параметра совпадает по синтаксису с инициализацией переменных. Но C++ допускает другой вид инициализации переменной:

```
тип имя(значение)
```

Такая форма присвоения значения для встроенных типов не допускается.

Параметры со значениями по умолчанию должны объявляться последними (крайними правыми) параметрами в списке. При вызове функции такие параметры можно не указывать — используется значение, заданное по умолчанию. Если же значение фактического параметра явно задано при вызове, то оно и используется.

Напишем функцию, выводящую на экран строку одинаковых символов. Такая функция часто бывает нужна при выводе на экран таблиц. Мы выводили строку минусов в программе вычисления полной таблицы умножения (см. листинг 1.17). Очевидно, функция должна иметь два параметра: выводимый символ и количество, причем, оба параметра можно сделать параметрами по умолчанию: символ — это обычно минус, а количество сделаем равным `60`. Текст функции приведен в листинге 2.6.

Листинг 2.6. Функция с параметрами по умолчанию

```
void repch(char ch = '-', int k = 60)
{ for (int i = 0; i < k; ++i) cout << ch; }
```

C++ разрешает такие вызовы этой функции:

```
repch();          //  выводится 60 минусов
repch('+');       //  выводится 60 плюсов
repch('=',52);    //  выводится 52 равно
```

Вызов функции с параметрами по умолчанию считается корректным, если не указываются самые правые параметры. Язык C++ запрещает писать вызовы с пропущенными первыми и заданными последними параметрами. Поэтому вызов

```
repch(,54);
```

писать нельзя. Аналогично при определении не допускается такой заголовок:

```
void repch(char ch = '-', int k)
```

К сожалению, без сообщений об ошибках проходит вызов:

```
repch(54);
```

Хотя очевидно, что предполагается вывод 54-х минусов, мы получим совершенно другой результат — на экране появится строка из 60 шестерок. Здесь мы сталкиваемся с неявным преобразованием типа: целое число 54 — это код символа `'6'`. Избежать подобной ошибки в данном случае невозможно, т. к. при передаче по значению обязательно выполняются неявные преобразования. Даже если мы не будем задавать значение первого параметра по умолчанию:

```
void repch(char ch, int k = 60)
```

вызов

```
repch(54);
```

все равно приведет к выводу на экран 60 шестерок.

Значения параметров по умолчанию можно задавать и в прототипах. Такое обычно бывает, если определение функции задано после вызова, например, как приведено в листинге 2.7.

Листинг 2.7. Параметры по умолчанию в прототипе

```
int f(int x = 3);              //  задаем значение по умолчанию
int main(void)
{  cout << f() << endl;
   return 0;
}
int f(int x) { return x; }     //  нельзя задавать значение по умолчанию
```

Тогда в определении умолчание задавать нельзя, иначе возникнет противоречие, и транслятор выдаст сообщение об ошибке.

Обычно в качестве значения по умолчанию используется константа, но допускается и выражение, приводимое к типу параметра. В частности, можно указывать вызов функции, как стандартной, так и определенной программистом (листинг 2.8).

Листинг 2.8. Вызов функции для вычисления значения по умолчанию

```
double g = 1, f;
double sqr(double x)            //  определение функции
{ return x * x; }
double ff(double x = sqr(f))    //  вызов функции для вычисления
                                //  значения
{ return x; }
int main(void)
{  f = 2;
   cout << "\n \n f =" << f << endl;
   cout << "\n ff(x = sqr(f) =" << ff();   //  вычисляется значение
                                          //  по умолчанию
   cout << "\n ff(3) =" << ff(3) << endl;  //  умолчание не используется
   return 0;
}
```

Данная программа совершенно корректно транслируется и выполняется, выводя на экран следующее:

```
f = 2
ff(x = sqr(f)) = 4
ff(3) = 3
```

Функция `ff` по умолчанию присваивает параметру `x` квадрат переменной `f`, поэтому во второй строке выводится значение `4`. Если параметр указан явно, то `ff` просто возвращает значение параметра. Этот пример наглядно демонстрирует, что параметры по умолчанию вычисляются при выполнении программы, а не при трансляции.

Описанный ранее фокус с несколькими выражениями в скобках (см. листинг 2.5) проходит и здесь (листинг 2.9).

Листинг 2.9. Несколько выражений в качестве значения по умолчанию

```
double g = 1, x;
double sqr(double x)                    //  определение функции
{ return x * x; }
```

```
double ff(double x = (++g, sqr(g)))    //  умолчание — вызов функции
{ return x; }
int main(void)
{  x = 6;
   cout << "\n ff(x = (++g, sqr(g)))=" << ff();   //  вычисляется
                                                  //  функция
   cout << "\n ff(x) =" << ff(3) << endl;         //  умолчание
                                                  //  не используется
   return 0;
}
```

В данном случае сначала вычисляется значение переменной `g`, затем это значение (`g = 2`) используется как параметр при вызове функции `sqr`. Именно значение, возвращаемое данной функцией, используется для присвоения начального значения параметра `x`. В качестве вызываемой в списке параметров функции можно указывать саму определяемую функцию (листинг 2.10).

Листинг 2.10. Вызов функции для собственного параметра по умолчанию

```
double y = 0;            //  определение глобальной переменной
double ff(double x);     //  прототип для вызова в списке параметров
double ff(double x = ff(y))    //  вызов с глобальной переменной
{return x;}              //  для присвоения значения по умолчанию
int main(void)
{  y = 6;
   cout << "\n ff(x = ff(y)) =" << ff();
   cout << "\n ff(3) =" << ff(3) << endl;
   return 0;
}
```

В программе определена функция `ff`, которая просто возвращает аргумент в качестве результата. Параметру присваивается значение по умолчанию, для чего вызывается та же самая функция. Поскольку в момент трансляции вызова в списке параметров функция `ff` еще не определена, требуется указать прототип, иначе программа не транслируется. Данная программа совершенно правильна и при выполнении выдает на экран следующее:

```
ff(x = ff(y)) = 6
ff(3) = 3
```

Вообще возможность вызова функции для присвоения начального значения параметра открывает широкие возможности. Например, можно вызывать функцию, которая вводит значение с клавиатуры или из файла. Таким обра-

зом, значение параметра по умолчанию будет изменяться от вызова к вызову! Это открывает широкие возможности для отладки. А если в этом нет необходимости, то при вызове функции параметр просто указывается явным образом.

Область видимости и "время жизни" переменных

Мы уже неоднократно упоминали о локальности или глобальности переменных. Разберемся в этом вопросе более детально. Любое имя в программе так или иначе объявляется. С момента объявления имя может использоваться в программе. *Область видимости имени* — часть программы, в которой имя можно использовать. Иногда вместо термина "область видимости" употребляется термин "область действия". С моей точки зрения, эти два термина эквивалентны, поэтому в дальнейшем мы будем использовать словосочетание "область видимости".

В C++ есть различные варианты области видимости имен, связанных с классами, файлами и пространствами имен, однако здесь мы рассмотрим традиционные правила, связанные с понятием функции. Такие правила области видимости имен определены в любом языке программирования, т. к. подпрограммы определены во всех без исключения языках.

"Время жизни" — это время существования переменной во время работы программы. Во всех наших примерах переменные были объявлены внутри тела функции. Однако в C++ переменные могут быть объявлены вне любой функции. Такие переменные называются *глобальными*. Их "время жизни" совпадает с временем работы программы. Имя глобальной переменной видно с момента объявления, т. е. в любой точке программы после точки объявления.

Все переменные, объявленные в теле функции, как и параметры, передаваемые по значению, являются *внутренними локальными переменными* функции. Это одно из проявлений принципа инкапсуляции, с которым мы будем иметь дело на протяжении всей книги. В данном случае это означает следующее:

- за пределами функции имена объявленных внутри нее переменных не видны, т. е. имя переменной нельзя использовать вне тела функции в других местах программы. Говорят, что область видимости локальных переменных — тело функции. Область видимости формальных параметров — также тело функции;
- до входа в функцию локальных переменных не существует. Они "рождаются" при входе в функцию и "умирают" при выходе из нее. Говорят, что "время жизни" локальных переменных ограничено временем выполнения

функции. То же относится к параметрам. Такая простая дисциплина реализуется с помощью стека[1]: локальные переменные и параметры автоматически размещаются в стеке программы при входе в функцию, а удаляются из стека при выходе из нее.

В связи с этим возникает три вопроса:

1. Можно ли внутри функции объявить переменную таким образом, чтобы она была видна вне тела функции?
2. Что происходит, если имя локальной переменной совпадает с именем глобальной (объявленной вне функции)?
3. Можно ли внутри функции объявить переменную таким образом, чтобы ее "время жизни" не было связано с временем выполнения функции?

Проще всего ответить на первый вопрос: внутри тела функции никаким образом нельзя объявить переменную так, чтобы она была видна вне тела функции.

Ответ на второй вопрос таков: внутренняя переменная "скрывает" внешнюю переменную внутри функции. Рассмотрим следующий пример:

```
int x = 5;        //  глобальная переменная
void f(void)
{  int x = 3;     //  локальная переменная
   cout << "x = << x << "— внутренний" << endl;
}
. . .
cout << "x =" << x << endl;
f();
cout << "x =" << x << endl;
. . .
```

При выполнении этого фрагмента программы на экран выведется:

```
x = 5
x = 3 — внутренний
x = 5
```

Это означает, что до вызова функции видна глобальная переменная `x`. При входе в функцию локальная переменная `x`, объявленная внутри функции `f()`, скрывает глобальную переменную — внутри функции глобальная переменная не видна. При выходе из функции локальная переменная `x` уничтожается и опять становится видна глобальная `x`.

---

[1] Стек — это динамическая структура данных, работающая по принципу Last In First Out (LIFO) — "последним пришел — первым ушел".

А если все-таки внутри функции требуется обращаться и к локальной, и к глобальной переменной, имена которых одинаковы? Тогда для доступа к глобальной переменной необходимо использовать операцию *разрешения контекста* ::. Следующий фрагмент программы поясняет, как это делается.

```
int x = 5;        //  глобальная переменная
void f(void)
{  int x = 3;     //  локальная переменная
   cout << "x = << x <<" — внутренний" << endl;
   cout << "x = << :: x <<" — глобальный" << endl;
}
int main(void)
{. . .
cout << "x =" << x << endl;
f();
cout << "x =" << x << endl;
. . .
}
```

При выполнении этого фрагмента программы на экран выведется:

```
x = 5
x = 3 — внутренний
x = 5 — глобальный
x = 5
```

Интересно, что принципу блочной видимости подчиняются и прототипы функций: прототип, объявленный внутри функции, вне функции не работает — он невидим. Более того, как и в случае переменных, если мы имеем два разных прототипа с одинаковым именем, один из которых вне функции, а второй внутри, то внутренний скрывает внешний. Вы спросите, могут ли два прототипа иметь одно и то же имя? C++ разрешает иметь функциям одинаковые имена, обеспечивая *перегрузку* функций.

То же самое касается и параметров по умолчанию в прототипе. Общее правило таково: в одной области видимости нельзя переопределять в прототипе (или в заголовке функции) параметр по умолчанию. Но в локальным прототипе это можно делать. Следующий простой пример демонстрирует это (его текст приведен в листинге 2.11).

Листинг 2.11. Переопределение прототипа локально

```
int f(int x = 3);
//  int f(int x = 7);       //  ошибка, если снять комментарий
int main(void)
{  cout << f() << endl;     //  выведет 3
```

```
   int f(int x = 9);        //  правильно
   cout << f() << endl;     //  выведет 9
   return 0;
}
int f(int x)                //  определение функции
{ return x; }
```

Пример демонстрирует и еще одну особенность локального прототипа: точно так же, как и локальная переменная, он начинает работать после объявления, а до того работает предыдущее видимое объявление.

Ответ на третий вопрос также положительный: внутри функции можно объявить локальную переменную, "время жизни" которой не зависит от времени выполнения функции. Для этого переменную надо объявить *статической*. Общий вид объявления статической переменной следующий:

```
static тип имя
```

Статическую переменную, так же как и любую другую, можно инициализировать при объявлении. Замечательное свойство таких переменных заключается в том, что они сохраняют свое значение по окончании работы функции. При очередном вызове используется оставшееся после предыдущей работы значение. В то же время статические переменные защищены от изменений извне.

Одним из традиционных применений статических переменных является использование их в качестве счетчиков для сбора различной статистической информации о вызове функции, например, о количестве вызовов. Это делается так:

```
unsigned int f(параметры)
{  static unsigned int count = 0;
       // . . .
   ++count;
   return count;
}
```

Вызов делается следующим образом:

```
insigned int n = f(параметры);
```

Эта информация может понадобиться для оптимизации программы.

Рассмотрим применение статических переменных для вычисления среднего арифметического значений, задаваемых пользователем с клавиатуры. Мы уже писали такую программу (см. листинг 1.11). В листинге 2.12 приведена реализация функции вычисления среднего с использованием статических переменных.

Листинг 2.12. Статические переменные в функции

```
double Average(double data)
{  static double summa = 0;     //  инициализация статических
   static int count = 0;        //  переменных при первом вызове
   count++;
   summa += data;
   return summa / count;        //  возврат нового среднего значения
}
```

Использование этой функции демонстрирует следующая программа:

```
int main()
{  double data = 1, middle;
   while (data)                 //  пока не задали ноль
   {  cout << "Введите число: ";
      cin >> data;
      middle = Average(data);   //  вычисление очередного среднего
      cout << "Среднее значение: " << middle << endl;
   }
   return 0;
}
```

При первом вызове сначала статические переменные `summa` и `count` получают значение `0`. Затем к значению переменной `summa` добавляется значение параметра `data`, а переменная `count` увеличивается на `1`. Выполняется возврат среднего (при первом вызове это значение переданного параметра). При втором вызове (и следующих) инициализация переменных не выполняется, а используются те значения, которые были вычислены во время работы в предыдущем вызове. Таким образом, функция действительно вычисляет среднее.

В C++ существует более локальная, чем блок, область видимости. Мы можем объявить переменную в скобках условного оператора `if`, например:

```
if (int y = 1) cout << y << endl;
```

Допускается и вторая форма инициализации:

```
if (int y(1)) cout << y << endl;
```

Область действия такой переменной — оператор `if`, причем как "ветка" `true`, так и "ветка" `false`. После завершения условного оператора попытки использования объявленной переменной приводят к ошибкам трансляции типа "необъявленная переменная". Естественно, в одном операторе может быть объявлено через запятую столько переменных, сколько требуется. Можно даже сделать их статическими — ошибки не будет.

Примечание

Система Borland C++ Builder 6, в отличие от Visual C++ 6, не позволяет это делать.

Аналогично можно объявлять переменную и в условии операторов цикла `while`. Область действия такой переменной — оператор цикла, после выхода из цикла эта переменная не определена. В операторе `for` тоже можно объявлять переменную в `выражение_1`. По стандарту она также должна быть локальной в пределах `for`, однако на практике эта переменная часто остается "жить" после цикла, и ее можно использовать далее в программе. Тем не менее рассчитывать в будущем на это не стоит, поскольку с каждой новой версией компиляторы все более строго придерживаются стандарта.

Примечание

Система Borland C++ Builder 6, в отличие от Visual C++ 6, выполняет это требование.

Передача параметров по ссылке

Передача параметра по значению — надежнейший способ, который, как мы теперь понимаем, практически всегда применяется для математических функций. Эти функции выдают единственный результат. Однако очень часто требуется получить от подпрограммы более одного результата, например, изменить значения нескольких переменных. Передача по значению в таких случаях не работает. Чтобы понять, в чем суть дела, рассмотрим классический пример: функцию, обменивающую значения двух целых переменных.

```
void Swap(int a, int b)
{ int t = a; a = b; b = t; }
```

Такая функция не вызовет протестов компилятора, но работать не будет, поскольку параметры передаются по значению. Пусть в программе объявлены две переменные `x` и `y`, и нам необходимо обменять их значения местами.

```
int x = 5, y = 3;
Swap(x, y);
cout << "x =" << x << ";
y = " << y << endl;
```

Оператор `cout` выведет на экран `x = 5` и `y = 3`. Таким образом, функция `Swap` не сделала свою работу. Вставим в функцию оператор вывода на экран:

```
void Swap(int a, int b)
{  int t = a; a = b; b = t;
```

```
   cout << "a = " << a << ";
   b = " << b << endl;
}
```

Повторив выполнение, увидим на экране следующее:

```
a = 3; b = 5
x = 5; y = 3
```

Таким образом, внутри функции обмен происходит. Тем не менее на значения переменных `x` и `y` это никак не влияет. Это происходит именно вследствие передачи параметров по значению: внутрь функции передаются только значения переменных. Никакой связи между передаваемыми значениями и переменными, в которых эти значения хранились вне функции, не сохраняется. Передаваемые значения помещаются в личные внутренние (локальные) переменные функции, и обмен происходит в этих внутренних переменных. Именно в таких случаях и необходим способ передачи параметров по ссылке. Определение функции `Swap` выглядит следующим образом (ее текст приведен в листинге 2.13).

Листинг 2.13. Функция обмена значений переменных

```
void Swap(int &a, int &b)        //  ссылки в заголовке
{ int t = a; a = b; b = t; }
```

Обращение к таким образом определенной функции не отличается от обращения при передаче параметров по значению:

```
Swap(x, y);
```

Выполнив приведенный выше пример, убеждаемся, что теперь все в порядке — переменные вне функции поменяли свои значения.

По стандарту языка C++ при передаче параметра по ссылке на месте фактического параметра можно указывать только переменную.

Примечание

Не все компиляторы транслируют такой способ передачи параметра в соответствии со стандартом. В компиляторах фирмы Borland по умолчанию включен режим расширений фирмы Borland, и проверка стандарта не выполняется. По умолчанию вызовы по ссылке фактически подменяются вызовами по значению, т. е. выражение вычисляется и передается в функцию. Поддержку стандарта можно включить, однако такая особенность интегрированной среды может спровоцировать возникновение труднообнаруживаемых ошибок.

Однако и в этом случае при вызове можно использовать операцию "запятая" для задания нескольких выражений на месте одного параметра (листинг 2.14).

Листинг 2.14. Несколько выражений на месте параметра-ссылки

```
double s(double &x)          //  параметр-ссылка
{ return (x++); }
int main(void)
{  double g = 1, f = 2;
   g = s((f = s(f), f));     //  несколько выражений на месте параметра
   cout << "\n \n f = " << f;
   cout << "\n g = " << g;
   return 0;
}
```

Программа совершенно корректна! Последним выражением в списке должна стоять переменная — параметр-то в функцию `s` передается по ссылке. Однако результат работы выглядит, на первый взгляд, неверно:

```
f = 3
g = 2
```

В самом деле, функция `s` вызывается дважды, а переменная `f` получает значение `3`, как будто функция работала только один раз. Но вспомним особенности возврата значения для выражения `x++`, уже рассмотренные нами. Так что все правильно.

Константные параметры функций

Так же как и переменные, параметры могут быть объявлены константными. Рассмотрим несколько вариантов заголовков одной и той же функции `f`.

```
int f(int x);           //  по значению
int f(int& x);          //  по ссылке
int f(const int x);     //  константный по значению
int f(const int& x);    //  константный по ссылке
```

Первый вариант передачи параметра позволяет нам в качестве фактического параметра задавать и переменные, и любые выражения. В теле функции мы можем использовать этот параметр как дополнительную локальную переменную. Мы не только получаем значение фактического параметра, но и можем присваивать новые значения этой переменной. Например, можно использовать такой параметр как счетчик цикла, изменяющийся на каждом шаге. Как мы знаем, никаких последствий для фактического параметра это не имеет.

Во втором случае мы имеем передачу параметра по ссылке, и можем в качестве фактического параметра задавать только переменную. Эта переменная похожа на глобальную — все изменения параметра внутри функции немед-

ленно отражаются на содержимом фактического параметра. Однако имя параметра локально в функции, поэтому конфликтов имен не происходит.

Третий вариант — передача по значению, но внутри тела функции запрещено изменять значение параметра: компилятор не будет транслировать программу при наличии изменяющих операторов в теле такой функции.

А вот четвертый вариант более интересен. Тут мы передаем параметр по ссылке, но запрещаем его изменение внутри тела функции: нельзя не только присвоить другое значение, но и ввести его с клавиатуры (хотя Visual C++ 6 выдает для этих случаев совершенно разные сообщения об ошибке).

При вызове такой функции разрешается задавать в качестве аргумента как переменную, так и произвольное выражение, чего нельзя было сделать при передаче параметра просто по ссылке (без слова `const`). Таким образом, этот способ аналогичен передаче по значению, но имеет два существенных преимущества по сравнению со старым способом:

- он значительно эффективнее — не происходит копирования значения;
- он явно самодокументирован.

Константность-неконстантность, как мы увидим далее, приобретает важное значение в случае параметров-массивов и параметров-контейнеров стандартной библиотеки.

Побочный эффект

В *предыдущем разделе* был рассмотрен способ передачи параметров по ссылке, который применяется в том случае, если требуется получить из функции не один, а несколько результатов. Говорят, что такая функция имеет побочный эффект. Изменение функцией нелокальных переменных называется *побочным эффектом*. Это явный и управляемый программистом побочный эффект. Гораздо хуже, когда вместо передачи нескольких параметров неопытный программист просто объявляет глобальные переменные и изменяет их непосредственно в теле функции. Подобная практика чревата ошибками. Может возникнуть такая ситуация, при которой значение выражения, содержащее вызов такой функции, будет зависимо от порядка операндов выражения. В некоторых случаях это противоречит здравому смыслу и общепринятым соглашениям. Например, операция умножения оказывается некоммутативной. Поэтому побочный эффект является потенциальным источником трудноуловимых ошибок в программе, и его следует избегать. Рассмотрим следующую программу[1] (текст которой приведен в листинге 2.15).

[1] Эта программа первоначально была написана на Pascal и приведена в книге: Зуев Е. А. Программирование на языке Turbo Pascal 6.0, 7.0. — М.: Веста, Радио и связь, 1993.

Листинг 2.15. Побочный эффект

```
int a, z;     //  нелокальные переменные
int C(int x)
{  z -= x;    //  изменение нелокальной переменной
   return (x * x);
}
void main(void)
{  z = 10;
   a = C(1);    //  "нормальный" вызов
   cout << z << ' ' << a << endl;    //  напечатается 9 и 1
   z = 10;
   a = C(z);    //  ОПАСНЫЙ вызов
   cout << z << ' ' << a << endl;    //  напечатается 0 и 100
   z = 10;
   a = C(10) * C(z);    //  ОЧЕНЬ ОПАСНЫЙ ВЫЗОВ
   cout << z << ' ' << a << endl;    //  напечатается 0 и 0
   z = 10;
   a = C(z) * C(10);    //  ОЧЕНЬ ОПАСНЫЙ ВЫЗОВ
   cout << z <<' ' << a << endl;     //  напечатается -10 и 10000
}
```

При первом вызове происходит изменение нелокальной переменной `z`, но там, где вызывается функция `C`, это изменение по тексту программы не видно. Такая ситуация обычно является источником ошибок, поскольку программисты часто забывают о побочном эффекте. Поэтому следует комментировать каждый вызов функции с побочным эффектом.

Результат второго вызова выглядит несколько парадоксально, но при некотором размышлении становится понятным. Передача `z` осуществляется по значению, поэтому в теле функции значение `z` сохранилось в локальной переменной `x`, в то время как значение нелокальной переменной `z` стало ноль.

Третий и четвертый вызовы демонстрируют зависимость результата от порядка вызова функции `C` в выражении. В этом случае достаточно трудно понять логику работы программы, поэтому таких ситуаций следует избегать. Тем более, что стандарт языка C++ не определяет порядка вычисления выражений: все оставляется на усмотрения разработчиков транслятора. Поэтому одна и та же программа с такими эффектами может в разных системах выдавать совершенно различные результаты.

Интересно, что в случае передачи параметра `x` по ссылке при трансляции в Visual C++ 6 следующей программы (текст которой приведен в листинге 2.16) результаты изменяются (`9` и `1`, `0` и `0`, `0` и `0`, `-10` и `0` соответственно). Как мы видим, и в этом случае значение переменной `z` зависит от порядка вызова функции.

Листинг 2.16. Побочный эффект со ссылками

```
#include <iostream.h>
int a, z;    //  глобальные переменные
int CC(int &x)
{  z -= x;
   return x * x;
}
int main(void)
{  int d;
   a = 1;
   z = 10;
   d = CC(a);
   cout << z << ' ' << d << endl;
   z = 10;
   d = CC(z);
   cout << z << ' ' << d << endl;
   a = 1;
   z = 10;
   d = CC(a) * CC(z);
   cout << z << ' ' << d << endl;
   a = 1;
   z = 10;
   d = CC(z) * CC(a);
   cout << z << ' ' << d << endl;
}
```

Вообще глобальные переменные часто создают проблемы вследствие своей излишней доступности. Недаром в объектно-ориентированном программировании особое значение придается инкапсуляции — сокрытию информации. Для иллюстрации возможных проблем рассмотрим простой пример, его текст приведен в листинге 2.17.

Листинг 2.17. Опасность использования глобальных переменных

```
int i;            //  глобальная переменная
double f()        //  функция, использующая глобальную переменную
{  double sum = 0;
   for (i = 0; i < 3; i++) sum += i;
   return sum;    //  возвращается правильное значение 3 = 0 + 1 + 2
}
int main(void)
{  double sum = 0;
```

```
    for (i = 0; i < 3; i++)     //  цикл выполняется один раз
    sum += f();
    cout << sum << endl;        //  выводится число 3
}
```

Эта простейшая программа не работает! Никаких сообщений компилятор, естественно, не выдает: все совершенно правильно с точки зрения синтаксиса и семантики языка C++. Тем не менее вместо правильного результата, равного `9`, программа выводит число `3`, как будто цикл в главной программе работал всего один раз. Так оно и есть на самом деле! Цикл в главной программе начинает работать при `i = 0`. Выполняется вызов функции `f()`, которая совершенно правильно выполняется, вычисляя сумму. Однако функция имеет побочный эффект: изменяет значение глобальной переменной `i`, которая является переменной цикла в вызывающей программе. Таким образом, при возврате из функции `f()` переменная цикла стала равна `3`, поэтому цикл сразу завершается, и локальная переменная `sum` получает значение `3`, а не `9`.

Если бы не была объявлена глобальная переменная, то компилятор сделал бы напоминание. Мораль, следующая из всего этого, такова — используйте глобальные переменные только в тех случаях, когда без них совершенно невозможно обойтись.

Неопределенное поведение не обязательно связано с глобальными переменными, оно может проявляться в некоторых конструкциях, на первый взгляд совершенно безобидных. В частности, порядок вычисления выражений-параметров функций по стандарту не определен. В качестве примера неопределенного поведения рассмотрим еще один пример. Пусть у нас определена функция:

```
int f(int &a, int &b)
{ return (a++ + b++); }
```

Определение совершенно корректно. Однако следующая последовательность операторов

```
int i = 1, j = 1;
j = f(j, j);
i = f(i, i);
cout << i << ';' << j << ';' << endl;
```

в системе Borland C++ Builder 6 выдает результат `3`, `3`, а Visual C++ 6 выводит на экран `4`, `4`. Таким образом, мы опять наблюдаем эффект многократного изменения одной и той же переменной в одном выражении. Таких выражений следует избегать. Например, в приведенной ранее функции мы

можем добиться совершенно определенного поведения, изменив выражение в операторе `return`:

```
return (a++, b++, a+b);
```

Тогда любой вызов

```
i = f(i, i);
```

в любой системе и в любом режиме трансляции выдает значение `6`.

Перегрузка функций

Как правило, разным функциям нужно давать разные имена. Но в некоторых случаях, когда ряд функций выполняют однотипную работу над объектами разных типов, гораздо удобнее иметь для всех функций одно-единственное имя. Использование одного имени для нескольких функций называется *перегрузкой* (overloading). Перегрузка функций — это одно из проявлений принципа *полиморфизма*, присущего объектно-ориентированным языкам.

Зайдем в справочник **Help** интегрированной среды Borland и попытаемся выяснить, какая функция вычисляет абсолютное значение своего аргумента. Оказывается, таких функций очень много — в зависимости от типа аргумента они по-разному называются:

```
int abs(int x);
long labs(long x);
double fabs(double x);
long double fabsl(long double x);
```

Такое количество разных функций, выполняющих по сути одну и ту же работу, не слишком удобно для программиста, который постоянно должен помнить, какую именно функцию можно использовать с тем или иным аргументом. Таким образом, перегрузку функций удобно применять там, где ряд функций выполняет однотипную работу над аргументами различных типов.

Основное правило перегрузки заключается в том, что перегруженными считаются функции с различными списками параметров. Ни тип возвращаемого значения, ни спецификация исключения не принимаются компилятором в расчет при определении перегруженных функций. Это означает, что такие, например, определения функций

```
void print(char t) { ... }
int print(char t) { ... }
```

Visual C++ 6 категорически отказывается транслировать и "обкладывает" данные определения аж двумя сообщениями об ошибках, а также советует

"разуть глаза" и взглянуть на определение функции `print` (see declaration of 'print'). Borland C++ поступает аналогично.

Таким образом, функции воспринимаются компилятором только в том случае, если у них различные списки параметров — либо по количеству, либо по типам. Тогда при вызове компилятор сначала ищет точное соответствие прототипа, а если не удалось — делает попытки сопоставить другие функции с таким же количеством параметров, надеясь на приведение типов параметров. Если по количеству параметров нет подходящих функций, то компилятор пытается "подставить" функции с параметрами по умолчанию или с переменным числом параметров, если такие определены. У Б. Страуструпа [37] правила поиска компилятором подходящей функции расписаны детально.

Реализуем перегруженный вариант функции, вычисляющей абсолютное значение своего аргумента:

```
int abs(int x);
long abs(long x) { return labs(x); }
double abs(double x) { return fabs(x); }
long double abs(long double x) { return fabsl(x); }
```

Таким образом, все функции имеют одно и то же имя, программисту не надо помнить лишней информации, а компилятор правильно разберется, какую функцию вызывать в зависимости от аргумента.

Однако при "перегрузках" возможны сюрпризы, вызываемые неоднозначностью в подборе функций. Причин этому может быть несколько. Например, одной из причин может быть способ передачи параметров. Пусть у нас есть два определения функций:

```
int f(int a) { ... }
int f(int &a) { ... }
```

Определения вопросов не вызывают, однако вызов — неоднозначен:

```
f(b);
```

Какая из двух функций вызвана? Компилятор Visual C++ 6 справедливо "возмущается", выдавая сообщение `C2668` ("двусмысленный вызов функции").

Использование слова `const` при объявлении перегруженных функций также имеет множество нюансов. В одних случаях программа нормально транслируется и при вызове функции компилятор находит нужную функцию. В других случаях программа транслируется нормально, а вызов неоднозначен. Есть варианты, при которых программа просто не транслируется.

В первом примере — передача параметров по значению.

```
void g(int a) {cout << "g(int a)" << endl;}
void g(const int a) { cout << "g(const int a)" << endl; }
```

Хотя определения формально разные, Visual C++ 6 выдает сообщение, что определение второй функции — "лишнее", тело прописано уже в первой. А второй заголовок воспринимается как прототип:

```
error C2084: function 'void __cdecl g(int)' already has a body
функция 'void __cdecl g(int)' уже имеет тело
```

Вариант со сылками,

```
void g(int &a) { cout << "g(int &a)" << endl; }
void g(const int &a) { cout << "g(const int &a)" << endl; }
```

напротив, не только транслируется, но и однозначно распознается при вызове: если в качестве аргумента задана константа, то вызывается функция со словом `const`; если же аргумент — переменная, то вызывается первая функция.

"Промежуточные" варианты:

```
void g(int &a) { cout << "g(int &a)" << endl; }
void g(const int a) { cout << "g(const int a)" << endl; }
```

или

```
void g(int a) { cout << "g(int a)" << endl; }
void g(const int &a) { cout << "g(const int &a)" << endl; }
```

транслируются нормально, но при вызове возникают проблемы.

Вызовы

```
g(2);
g(v);
```

приводят к выдаче сообщению `C2668` о двусмысленности вызова функции `g`.

Неоднозначность может быть связана с использованием параметров по умолчанию:

```
void f(int a)                  //  один параметр целого типа
{ cout << "f(a)" << endl; }
  void f(int a, int b = 0)     //  один или два параметра
{ cout << "f(a, b = 0)" << endl; }
```

Компилятор ничего не имеет против наличия таких определений в вашей программе. Однако попытка вызвать функцию с одним целым аргументом опять приводит к ошибке трансляции `C2668`. Вызов с двумя аргументами, естественно, корректно срабатывает. Таким образом, параметр по умолчанию никогда не может быть использован. Интегрированная среда Borland ведет себя аналогичным образом.

Добавим в программу определение функции с одним параметром другого типа:

```
void f(double a)              //  один параметр
{ cout << "f(double a)" << endl; }
```

Теперь вызов с одним аргументом проходит, но аргумент обязательно должен быть не целого типа. Для вызова с одним аргументом целого типа (причем, не принимается не только `int`, но и типы `long` и `short`) компилятор по-прежнему выдает сообщение об ошибке. Если мы уберем из программы определение функции с одним целым параметром, тогда появляется возможность использовать вызов с параметром по умолчанию: с целым аргументом вызывается функция с двумя параметрами, с дробным аргументом — функция с одним параметром.

Мы еще не раз столкнемся с проблемами неоднозначности при перегрузке. Все эти проблемы связаны в большинстве случаев с неявным преобразованием.

Шаблоны функций

Естественным развитием механизма перегрузки является механизм шаблонов. При перегрузке функций приходится писать практически одно и то же несколько раз. Например, хотелось бы, чтобы функция обмена значений двух переменных "умела" это делать для любых встроенных типов. Но тогда нам придется написать 2 варианта функции для обмена символьных переменных (для `signed char` и `unsigned char`), 6 вариантов для работы с целыми и 3 варианта для обмена дробных чисел. Итого — 11 практически одинаковых функций. А если у нас в программе определены еще и новые типы, то для каждого из них придется добавлять свою функцию. Это несколько утомительно, поэтому чревато ошибками. Функция обмена проста, а в сложных функциях довольно легко допустить ошибку.

Поэтому хотелось бы написать некий обобщенный вариант один раз и потом использовать его, уже не заботясь о типах. Видимо, Б. Страуструп тоже не любит писать несколько раз одно и то же, поэтому он и добавил в C++ шаблоны, чтобы можно было не писать "гроздья" одних и тех же функций, отличающихся только типами параметров. Общий вид *шаблона функции* выглядит следующим образом:

```
template <список параметров шаблона>
заголовок функции
тело функции
```

Сначала напишем конкретный шаблон функции обмена (его текст приведен в листинге 2.18), а потом разберем его особенности.

Листинг 2.18. Шаблон функции обмена

```
template <class T>
void swap(T& x, T& y)
{ T z = x; x = y; y = z; }
```

Итак, мы видим, что в списке параметров шаблона определен параметр `T`, который использован как имя типа в заголовке и в теле функции. Имея такой шаблон мы можем больше не заботиться о конкретных типах параметров. Причем, в отличие от механизма перегрузки, использование шаблона сокращает не только исходный текст, но и исполняемый модуль. Дело в том, что компилятор "не обращает внимания" на шаблон до тех пор, пока не обнаружит в тексте вызов функции с конкретными типами данных. Допустим, у нас в программе есть такие строки:

```
long k = 7, d = 12;
swap(k, d);
float p = 62.44, q = 68.3;
swap(p, q);
```

Компилятор сформирует только два определения:

```
void swap(long& x, long& y) { long z = x; x = y; y = z; }
void swap(float& x, float& y) { float z = x; x = y; y = z; }
```

Остальных девяти вариантов функции, которые нам пришлось бы писать при перегрузке, создано не будет. Процесс конкретизации шаблона называется *инстанцированием*. Вообще-то инстанцирование можно указывать и явным образом

```
swap<long>(k, d);
swap<float>(p, q);
```

но в данном случае в этом нет необходимости, поскольку компилятор самостоятельно выводит тип аргументов. Однако в более сложных случаях явное указание бывает необходимо.

Как мы видим, в заголовке и теле функции имена параметров шаблона обычно применяются в качестве имени типа. Каждый формальный параметр шаблона, который используется как имя типа, обозначается служебным словом `class`, за которым следует имя параметра (идентификатор).

Примечание

В стандарте определено и другое слово `typename`, которое можно использовать при написании формального параметра шаблона. Но эта особенность не реализована в старых компиляторах вроде Borland C++ 3.1, однако в более поздних реализациях — Borland C++ 5, Borland C++ Builder 6 — она уже присутствует.

Стандарт C++ разрешает в качестве параметров шаблона задавать и обычные параметры, по форме совпадающие с объявлением переменной.

Примечание

В Borland C++ 3.1 эта возможность не была реализована, т. к. появилась позже.

Например, приведенная ранее функция вывода последовательности одинаковых символов (см. листинг 2.6) может быть записана в виде шаблона, текст которого приведен в листинге 2.19. Тогда нет необходимости задавать второй параметр в функции.

Листинг 2.19. Шаблон с параметром целого типа

```
template <int n>
void repch(char ch = '-')
{ for (int i = 0; i < n; ++i) cout << ch; }
```

При вызове вместо второго параметра надо в угловых скобках явно прописывать константное выражение. Вызов такой функции может быть следующим:

```
repch<10>('+');
```

Однако параметр шаблона — это все-таки не параметр функции. Например, его нельзя задать по умолчанию — выдается сообщение об ошибке.

```
template <int n = 10>
void repch(char ch = '-')
{ for (int i = 0; i < n; ++i) cout << ch; }
```

Стандарт C++ допускает перегрузку шаблонов и функций. Пусть, например, у нас в программе определены и шаблон (листинг 2.19), и одноименная функция с параметрами по умолчанию (см. листинг 2.6). Тогда допускаются следующие вызовы шаблона и функции:

```
repch<10>('=');
repch('=');
repch('+', 30);
repch<5>('+');
```

Примечание

В системе Visual C++ 6 очень плохо реализованы шаблоны — много ошибок и плохо поддерживается стандарт. В частности, приведенные примеры работают не все. В системе Borland C++ Builder 6 все работает правильно.

На этом пока закончим обсуждение шаблонов, хотя в данной теме еще очень много нюансов. Шаблоны — очень мощный инструмент, что и демонстрирует стандартная библиотека STL.

Символы

Совершим первый экскурс в проблемы взаимодействия C++ и операционной системы. Рассмотрим ввод и обработку русских букв. В состав стандартных библиотек C++ включена библиотека ctype.h, которая осталась в наследство от C. В эту библиотеку входит ряд функций, определяющих тип своего аргумента — символа. Большинство из них перечислены в табл. 2.1.

Таблица 2.1. *Функции библиотеки ctype.h*

Прототип	Назначение функции
`int isdigit(int c)`	Является ли аргумент цифрой `0-9`
`int isxdigit(int c)`	Является ли аргумент шестнадцатеричной цифрой `0-9`, `a-f`, `A-F`
`int isalpha(int c)`	Является ли аргумент буквой `a-z`, `A-Z`
`int isalnum(int c)`	Является ли аргумент буквой или цифрой
`int islower(int c)`	Является ли аргумент маленькой буквой `a-z`
`int isupper(int c)`	Является ли аргумент большой буквой `A-Z`
`int isspace(int c)`	Является ли аргумент пробельным символом
`int isgraph(int c)`	Является ли аргумент видимым символом
`int tolower(int ch)`	Если аргумент `A-Z`, то результат `a-z`
`int toupper(int ch)`	Если аргумент `a-z`, то результат `A-Z`

Функции, имеющие префикс `is`, являются *функциями-предикатами.*

Функция-предикат — это функция, результатом работы которой является логическое значение "истина" или "ложь". Но поскольку библиотека стандартизирована еще в C, функции возвращают не логические значения `true` и `false`, а целые. Но если результат "ложь", как обычно, равен `0`, то результат "истина" отнюдь не равен `1`. В справочной системе Borland C++ 3.1 указано, что функции возвращают ненулевое значение. И действительно, операторы

```
cout << isalpha('a') << endl;
cout << isdigit('0') << endl;
```

выводят на экран `8` и `2` соответственно. В системе Visual C++ 6 те же функции выводят `2` и `4`. Поэтому полагаться на то, что возвращаемое значение будет равно `1`, нельзя. Исключение составляют функции `tolower` и `toupper`. Первая преобразует аргумент — большую английскую букву — в соответствующую маленькую, а вторая наоборот.

Все эти функции чрезвычайно полезны при обработке символов и строк, однако они работают только с английским алфавитом. Нам, естественно, хотелось бы иметь обработку и русских букв. Для этого нам надо знать коды русских букв. Однако следует напомнить, что представление национальных алфавитов является наиболее системно-зависимым.

В состав всех систем фирмы Borland входит библиотека conio.h, которая содержит очень много полезных функций. Такая же библиотека есть и в составе системы Visual C++ 6. В первую очередь нам потребуется функция ввода символов, прототип которой выглядит так:

```
int getch(void);
```

Эта функция позволяет получить с клавиатуры любую клавишу, даже управляющие клавиши курсора и функциональные. Сначала напишем программу, выполняющую ввод клавиши и выводящую код клавиши на экран. Программа должна работать, пока мы не нажмем клавишу <Esc>, код которой равен 27. Чтобы выводились положительные коды, надо прописать переменную для вводимого символа как `unsigned char`. Программа определения кодов символов приведена в листинге 2.20.

Листинг 2.20. Определение кодов символов

```
#include <iostream.h>
#include <conio.h>
int main()
{  const int Esc = 27;       //  ASCII-код ESC
   unsigned char ch = 0;     //  нужен беззнаковый char
   while (ch != Esc)
  {  ch = getch();           //  ввод кода — целого в char
     cout << ch << '='
     << int(ch) << endl;     //  вывод кода
  }
   return 0;
}
```

Выясняется, что большие русские буквы имеют последовательные коды в диапазоне от 128 до 159. Исключение составляет только буква "Ё", которая имеет код 240. А вот с маленькими буквами дело обстоит не так хорошо: буквы от "а" до "п" имеют коды от 160 до 175 (отличаются от больших букв ровно на 32), а вот буквы от "р" до "я" кодируются числами от 224 до 239. Между "п" и "р" разрыв в 48 символов: коды в диапазоне 176—223 соответствуют символам псевдографики текстового режима работы дисплея. Кроме того, как и в случае больших букв, исключением является буква "е", которая имеет код 241.

Примечание

В системах Visual C++ 6 и Borland C++ Builder 6 эта программа работает точно так же.

Теперь, когда мы разобрались с кодами, можно заняться написанием функций, аналогичных функциям библиотеки ctype.h. Нас интересуют функции `isalpha`, `isalnum`, `isgraph`, `islower`, `isupper`, `tolower`, `toupper`. Текст программы русификации предиката `isupper` приведен в листинге 2.21.

Листинг 2.21. Русификация `isupper`

```
int isrupper(int ch)
{  if ((127 < ch) && (ch < 160) || (ch == 240)) return 1;
   else return isupper(ch);
}
```

Функция возвращает `1`, если аргумент — большая русская буква. Мы вызвали стандартную функцию внутри нашей, поэтому имя нашей функции необходимо изменить. Вместо вызова стандартной функции можно было бы явно прописать проверку для больших английских букв,

```
if ((64 < ch) && (ch < 91)) return 1;
```

однако в этом случае мы не гарантируем возврат стандартного значения. Лучше всегда использовать стандартные функции, а не пытаться имитировать их работу.

Теперь напишем соответствующую функцию для проверки на "малость" русских букв. Она совершенно аналогична, однако надо учесть "разрыв" кодов. Текст программы русификации предиката `islower` приведен в листинге 2.22.

Листинг 2.22. Русификация `islower`

```
int isrlower(int ch)
{  if ((159 < ch) && (ch < 176) || (223 < ch)
                  && (ch < 240) || (ch == 241))
   return 1;
   else return islower(ch);
}
```

Теперь эти функции мы можем использовать для реализации других функций.

```
int isralpha(int ch)
{ return (isrlower(ch) || isrupper(ch)); }
```

Понятно, что `isalnum` и `isgraph` реализуются аналогично. А вот реализация функций-преобразователей несколько интереснее. Текст программы русификации предиката `tolower` приведен в листинге 2.23.

Листинг 2.23. Русификация `tolower`

```
unsigned char rtolower(int ch)
{  if ((127 < ch) && (ch < 144))        return (ch += 32);
   else if ((143 < ch) && (ch < 176)) return (ch += 80);
   else if (ch == 240)                return ++ch;
   else tolower(ch);
}
```

Как видите, нам опять потребовалось явно использовать диапазоны кодов русских букв. Кроме того, обратите внимание на то, что мы вынуждены изменить тип возвращаемого значения, поскольку иначе выводятся целые числа. Текст программы русификации предиката `toupper` приведен в листинге 2.24.

Листинг 2.24. Русификация `toupper`

```
unsigned char rtoupper(int ch)
{  if ((159 < ch) && (ch < 176))      return (ch -= 32);
   else if ((223 < ch) && (ch < 240)) return (ch -= 80);
   else if (ch == 241)                return --ch;
   else toupper(ch);
  }
```

Вызывает интерес тот факт, что наши функции в среде Borland C++ 3.1 работают правильно и с константами и с переменными. Например, мы выполним следующую последовательность операторов:

```
ch = getch();
cout << rtolower(ch) << endl;
cout << rtolower('A') << endl;
```

И первая строка при вводе символа `A`, и вторая сработают одинаково: на экране появится маленькая буква `a`. А вот в системах, работающих под Windows, для переменной получается правильный результат, а для константы — совершенно другой. Дело здесь в кодировке русских символов, которые мы набираем в окне редактора интегрированной среды — редактор-то работает в Windows, а выполняется программа в окне DOS! Мы еще вернемся к этой проблеме позднее.

Макросы и inline-функции

Все наши функции — очень небольшого объема. Традиционно в C для реализации таких функций использовался препроцессор. Препроцессор — это "древняя" часть систем C++, он появился вместе с C и продолжает "жить" до сих пор. Однако в C++ его основное назначение — управление организацией больших программ и *условная компиляция*. Препроцессор — это программа, выполняющая предварительную обработку текста программы. Результат работы препроцессора поступает на вход компилятору.

Примечание

Препроцессор можно увидеть, например, в каталоге Bin интегрированной среды Borland C++ Builder 6 — это программа cpp32.exe.

Конструкции препроцессора называются директивами. Мы уже знаем одну из директив препроцессора `#include` *(см. гл. 1)*. Самой широко применяемой директивой препроцессора является, вне всякого сомнения, директива `#define`. Простейший формат ее таков:

```
#define имя строка
```

С ее помощью логические операции можно заменить английскими словами:

```
#define and &&
#define or ||
#define not !
```

В C эта директива использовалась для определения констант. Достаточно заглянуть в системные файлы float.h или limit.h, чтобы понять, в чем дело. Например, минимальные и максимальные значения типа `float` и `double` в системе Visual C++ 6 определены во float.h так:

```
#define DBL_MAX     1.7976931348623158e+308 /* max value */
#define DBL_MIN     2.2250738585072014e-308 /* min positive value */
#define FLT_MAX     3.402823466e+38F    /* max value */
#define FLT_MIN     1.175494351e-38F    /* min positive value */
```

В файле limit.h находятся аналогичные определения для целых чисел, например:

```
#define INT_MIN   (-2147483647 - 1) /* minimum (signed) int value */
#define INT_MAX    2147483647  /* maximum (signed) int value */
#define UINT_MAX   0xffffffff  /* maximum unsigned int value */
```

Огромное количество констант определено таким образом в операционной системе Windows. Все эти определения работают однотипно: при подключении соответствующего файла с помощью директивы `#include` мы получаем

возможность использовать в своей программе определенные таким образом имена.

Примечание

По сложившейся традиции имена, определяемые в `#define`, желательно писать только большими английскими буквами.

Препроцессор, обрабатывая текст программы, заменяет определенные таким образом имена на соответствующую константу-строку. Но мы знаем, что в C++ для объявления констант уже появился модификатор `const`, с помощью которого и объявляются константы. Б. Страуструп настоятельно рекомендует использовать `const` и не применять для определения констант `#define`.

Имя, определяемое в директиве `#define`, называется *макросом*, а строка, заменяющая имя при обработке, *телом макроса*. Препроцессор позволяет определить макрос с параметрами. Рассмотрим ранее определенную нами функцию `max3` (см. листинг 2.3). Определение функции очень короткое, однако компилятор не различает, короткая или длинная функция, и формирует стандартную последовательность команд процессора для вызова функции. Учитывая, что параметров целых три, после трансляции вызов функции может оказаться длиннее самой функции. Таким образом, при каждом вызове мы имеем накладные расходы. В C препроцессор использовался как раз для того, чтобы эти накладные расходы уменьшить. Если мы определим не функцию, а макрос с параметрами,

```
#define max3(a, b, c) max(max(a, b),c)
```

то при обработке текста программы препроцессор на место имени макроса

```
int x = max3(c + d, 5, Cube(n));
```

подставит тело макроса, заменив параметры соответствующим образом:

```
int x = max(max(c + d, 5), Cube(n));
```

В файлах, размещенных в системном каталоге include интегрированной среды, можно найти массу примеров подобного применения препроцессора. Например, мы использовали функции для получения случайных чисел `randomize()` и `random()`. Открыв файл stdlib.h, мы обнаруживаем, что на самом деле это не функции, а макросы с параметрами:

```
#define RAND_MAX 0x7FFFU    //  шестнадцатеричная константа
#define random(num)(int)(((long)rand() * (num)) / (RAND_MAX + 1))
#define randomize()srand((unsigned) time(NULL))
```

То же самое можно сказать и о функциях `max` и `min`:

```
#define max(a, b)  (((a) > (b)) ? (a) : (b))
#define min(a, b)  (((a) < (b)) ? (a) : (b))
```

Однако на этом пути оптимизации нас подстерегает множество "подводных камней". Препроцессор не выполняет никаких проверок, а просто заменяет один текст другим. Именно поэтому при последующей трансляции могут возникать малопонятные ошибки. Пусть, например, определение `max` записано без скобок.

```
#define max(a, b) a > b ? a : b
```

Тогда вызов

```
int d = max(a += 1, b += 2);
```

раскроется в текст:

```
int d = a += 1 > b += 2 ? a += 1 : b += 2;
```

Такой текст является неправильным, т. к. приоритет операции `>` больше приоритета `+=`. Сначала выполнится выражение `1 > b`, а при попытке выполнения операции `+=`, расположенной справа, возникнет ошибка типа "слева от присваивания должна быть переменная".

Чтобы избавиться от подобных проблем, и в то же время избавиться от накладных расходов на вызов функции, в C++ добавили *inline-функции*. Короткие функции теперь можно объявлять с зарезервированным словом `inline`. Определение нашей функции `max` тогда будет выглядеть так:

```
inline int max(int a, int b) { return (a > b ? a: b); }
```

Эта конструкция проверяется компилятором во время трансляции, поэтому непредсказуемых ошибок при выполнении быть просто не может. В то же время ключевое слово `inline` сообщает компилятору, что он может вместо стандартной последовательности вызова функции подставлять непосредственно тело функции, модифицируя его параметрами. К сожалению, важный вопрос подстановки оставлен на усмотрение компилятора, так что нет никакой гарантии, что даже при вашем настойчивом желании компилятор создаст подставляемую функцию. У него найдется масса причин отказаться это делать, например:

- функция слишком велика; причем в стандарте никак не объясняется слово "слишком". Таким образом, все зависит от компилятора;
- функция содержит цикл `switch` или `goto`;
- функция вызывается более одного раза в выражении.

Есть и еще причины, но и этого хватит. Так что нам остается лишь вежливо "попросить" компилятор с помощью слова `inline`, а уж как он решит — это дело его разработчика.

Применение препроцессора с пользой

Вернемся к препроцессору. Препроцессор предоставляет программисту пару операций и ряд *встроенных макросов*, которые могут с большой пользой ис-

пользоваться для диагностики ошибочных ситуаций. Подробно мы их рассматривать не будем, остановимся только на операции # (решетка). Ее действие проясняет следующий макрос:

```
#define DIAG(v) cout << #v << '=' << v << endl
```

Этот простой макрос часто бывает полезен при отладке программы. Пусть некоторая переменная tt содержит значение 55.4. Следующий оператор

```
DIAG(tt);
```

выведет на экран:

```
tt = 55.4;
```

Таким образом, мы получаем документированный вывод, значительно облегчающий отслеживание изменений переменных по ходу работы программы.

Так как препроцессор просто выполняет подстановку текста, то в качестве аргумента можно задавать целый оператор. Кроме того, можно использовать стандартные встроенные макросы __LINE__, __FILE__, __DATE__ и ряд других. Пример использования встроенных макросов приведен в листинге 2.25.

Листинг 2.25. Макрос с параметром-оператором

```
#define Print(arg) \
cout << #arg << ';' << endl; \
cout <<__DATE__ << ';' \
cout <<__FILE__ << ';' << endl \ << __LINE__ << endl; \
arg    //  должен быть оператор
```

Обратная косая черта в конце каждой строки "склеивает" предыдущую строку со следующей. Нам без этого не обойтись, т. к. препроцессор считает телом макроса строку до ближайшего конца строки (или до комментария). Обратная косая черта служит признаком того, что тело макроса продолжается на следующей строке.

Мы написали аргумент на отдельной строке и без точки с запятой — чтобы их не было две в результате. И мы использовали три встроенных макроса: __DATE__ нам покажет дату компиляции, __FILE__ — какой файл транслируется, а __LINE__ — в какой строке был вызван макрос Print (его текст приведен в листинге 2.26).

Листинг 2.26. Программа с макросом Print

```
#include <iostream.h>
    //  комментарии тоже считаются
    //  за отдельную строку
```

```
#define Print(arg) \
cout << #arg << ';' <<endl; \
cout <<__DATE__ << ';' << endl; \
cout <<__FILE__ << ';' << endl \
     <<__LINE__ << endl; \
arg
int main()
{  Print(int a = 3);    //  11-я строка
   cout << a << endl;
   return 0;
}
```

К сожалению, препроцессор считает все строки исходной программы, даже комментарии и вообще пустые, поэтому эта программа выведет на экран:

```
int a = 3;       //  это наш arg
May 25 2003;     //  это __DATE__
PREPROC.CPP;     //  это __FILE__
11               //  и __LINE__
3                //  это оператор вывода в main
```

На этом закончим пока обзор возможностей препроцессора. Мы вернемся к нему при обсуждении организации многомодульных программ *(см. гл. 11)*.

В состав любой интегрированной системы входит специальный отладочный макрос `assert`, который позволяет проверять выражения во время выполнения программы. Его прототип прописан в заголовке `assert.h`. В *гл. 1* мы написали программу вывода таблицы умножения с проверкой множителя (см. листинг 1.3). Вместо оператора `if` мы могли бы применить макрос `assert` (его текст приведен в листинге 2.27).

Листинг 2.27. Таблица умножения с проверкой множителя макросом `assert`

```
#include <assert.h>
#include <iostream.h>
int main()
{  int k;        //  объявление переменной
   cout << "Введите множитель от 1 до 9: ";
   cin >> k;     //  ввод числа
   assert ((0 < k) && (k < 10));    //  внешние скобки обязательны
   cout << k << "* 1 =" << k * 1 << endl;
   cout << k << "* 2 =" << k * 2 << endl;
   cout << k << "* 3 =" << k * 3 << endl;
   cout << k << "* 4 =" << k * 4 << endl;
```

```
    cout << k << "* 5 =" << k * 5 << endl;
    cout << k << "* 6 =" << k * 6 << endl;
    cout << k << "* 7 =" << k * 7 << endl;
    cout << k << "* 8 =" << k * 8 << endl;
    cout << k << "* 9 =" << k * 9 << endl;
    cout << k << "* 10 =" << k * 10 << endl;
    return 0;
}
```

Если мы зададим значение переменной `k` вне проверяемого интервала, то программа закончится аварийно. При этом на экран будет выведено сообщение об аварийном завершении программы: `Assertion failed`. В сообщение включаются:

- проверяемое выражение непосредственно в том виде, как записано в программе. В нашем случае это выражение `(0 < k) && (k < 10)`;
- имя файла, в котором произошла "авария". Если мы записали файл с именем `cassert`, то на экране будет выведено `file cassert.cpp`;
- номер строки программы, в которой произошла авария. В данном случае это будет `line 6`.

Примечание

Системы Borland C++ 5, Borland C++ Builder 6, Visual C++ 6, работающие в Windows, выводят полное имя файла.

Как видите, макрос выдает практически то же самое, что и наш макрос `Print`. Макрос, однако, не зря называется отладочным, т. к. аварийное завершение программы неприемлемо в реальных программах: лучше вежливо сообщить пользователю об ошибке и попросить его задать данные снова. Но этот макрос чрезвычайно полезен при проверке значений параметров функции во время отладки. Макрос можно отключить, если прописать перед директивой

```
#include <assert.h>
```

директиву

```
#define NDEBUG
#include <assert.h>
```

В этом случае никаких аварийных завершений программы не будет, а вызовы `assert` будут восприниматься как комментарий.

ГЛАВА 3

Группы данных

Все элементы данных, которые мы рассматривали до сих пор, являлись "одиночными", или, как принято в программировании их называть, *скалярными*. Между тем, большинство реальных задач должны обрабатывать группы данных, как правило, довольно большого объема. Классическим примером является задача начисления заработной платы. Во-первых, для одного работника требуется множество *разнородных* данных: фамилия, имя, отчество, квалификация, оклад, разряд, количество отработанных дней и т. д. Все эти данные по работнику желательно объединить в одну группу (в полном соответствии с принципом инкапсуляции информации). Во-вторых, эта задача решается для множества людей — работников предприятия. В этом случае мы оперируем группой *однородных* данных.

Уже в самых первых языках программирования такие конструкции были реализованы. Группа однородных данных называется *массивом* — это название принято во всех без исключения языках программирования. Последнее время в C++ в связи с включением в стандарт библиотеки шаблонов используется и другое название — *контейнер*. Но контейнеры — это более общие структуры, реализованные уже средствами самого языка C++. А массивы унаследованы от C.

Конструкция группировки разнородных данных в разных языках программирования называется по-разному. Например, в языке Pascal такая конструкция именуется *запись*. В C++ (по наследству от C) принято название *структура*.

Массивы

Чтобы разобраться, что такое массивы и зачем они нужны, давайте напишем программу, которая будет вычислять, какой день недели приходится на число и месяц текущего года (2003). Очевидно, нам потребуется перемен-

ная, содержащая месяц. Пока будем представлять месяц целым числом: январь — 1, февраль — 2 и т. д. Потребуется также и переменная, в которой будет находиться число месяца. Кроме того, нам потребуются 12 чисел, представляющих количество дней в месяце: в январе — 31 день, в феврале — 28 дней и т. д. до декабря. Алгоритм очевиден: необходимо сложить количество дней всех месяцев до заданного, прибавить заданное число дней и получить остаток от деления на 7. Кроме того, мы должны учесть, с какого дня недели начинается текущий год (со среды) и добавить количество дней оставшихся в прошлом году (два — понедельник и вторник). Это и будет искомый день недели.

Вроде бы все просто. Но нам требуется написать универсальную программу, которая вычисляет день недели для любого числа 2003 года. И тут нас поджидают проблемы с вычислением суммы дней месяцев до заданного — мы должны предусмотреть 12 вариантов получения суммы. Если месяц первый, то сумма равна 0. Для каждого следующего месяца мы должны добавлять количество дней предыдущего. Мы можем написать получение суммы с помощью оператора switch:

```
switch (месяц) {
case 1: сумма = 0;                          break;
case 2: сумма = 31;                         break;
case 3: сумма = 31 + 28;                    break;
 ...
case 12: сумма = 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30;
                                            break;
}
сумма += день;
```

Однако сразу становится понятно: мы имеем возможность использовать такую схему только потому, что вариантов мало — всего 12. Но большинство программ должны выполнять гораздо большее количество аналогичных вычислений. Поэтому нам требуется таким образом организовать константы, чтобы можно было применить оператор цикла. Именно в случаях, аналогичных данному, используется массив. Данные однородны, поскольку все они имеют один и тот же тип — в нашем случае целый. Так же, как и простую переменную, массив необходимо объявить:

```
тип имя[количество];
```

В этой записи количество определяет, сколько элементов заданного типа может поместиться в массиве. В нашей задаче мы должны объявить массив из 12 элементов, соответствующих нашим месяцам:

```
int month[12];
```

Для задания количества допускается использовать любое константное выражение. Это означает, что C++ не разрешает задавать в качестве количест-

ва ни переменные, ни выражения, которые вычисляются во время работы программы.

Так же, как простую переменную, массив можно инициализировать. Объявим наш массив месяцев с инициализацией таким образом:

```
int month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

При инициализации массива разрешается не указывать количество элементов:

```
int month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

В этом случае количество элементов определяется количеством констант. Если задано количество, а констант меньше (больше задать нельзя — программа не транслируется), то оставшиеся элементы массива заполняются нулями. Таким образом, обнулить массив с заданным количеством элементов можно очень просто:

```
int v[10] = {0};
```

Лучше количество элементов задавать, т. к. это убережет от случайной ошибки, когда количество констант окажется больше положенного. Кстати, конструкция

```
int v[] = {0};
```

обозначает массив из одного элемента, и этот элемент равен нулю.

Если количество элементов не задано, то мы можем узнать как размер массива, так и количество элементов в нем, используя известную нам операцию `sizeof` — надо размер массива разделить на размер элемента. Следующий оператор выведет на экран числа `24` и `12` — размер массива в памяти и количество элементов в массиве:

```
cout << sizeof(month) << sizeof(month) / sizeof(int) << endl;
```

Каждый элемент массива имеет определенный номер, который в программировании обычно называют *индексом*. В C++ принято (по наследству от C) нумерацию элементов в массиве начинать с нуля. Чтобы использовать отдельный элемент массива в программе, мы должны написать имя и после него — индекс элемента в квадратных скобках. Причем, в выражениях можно использовать такую конструкцию как справа от знака присваивания, так и слева. Таким образом, в нашем случае `month[0]` соответствует январю, `month[1]` — февралю и т. д. до декабря (`month[11]`).

На месте индекса разрешается писать выражение, которое вычисляется во время работы программы. Это позволяет использовать с массивами оператор цикла со счетчиком `for`. Например, следующий цикл выведет на экран в столбик все наши константы из массива `month`:

```
for (int i = 0; i < 12; ++i) cout << month[i] << endl;
```

Аналогично массив можно заполнить, вводя элементы массива с клавиатуры:

```
for (int i = 0; i < 12; ++i) cin >> month[i];
```

Для отладки мы можем заполнить массив случайными числами, используя функции генерации случайных чисел:

```
for (int i = 0; i < 12; ++i) month[i] = random(100);
```

Операция присваивания с массивами не работает:

```
int v[10];
v = { 1, 2, 3, 4, 5 };
```

Хотя она очень похожа на инициализацию, однако компилятор "ругается" и не пропускает такую конструкцию. Присвоить один массив другому можно только поэлементно, используя тот же цикл `for`. Пусть у нас объявлены два массива `a` и `b` размером `20` целых элементов. Присвоить массив `b` массиву `a` можно так:

```
for (int i = 0; i < 20; ++i) a[i] = b[i];
```

Теперь мы можем написать первую версию нашей программы, текст которой приведен в листинге 3.1.

Листинг 3.1. Вычисление дня недели

```
#include <iostream.h>
int main()
{  unsigned int day, m;
   int month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
   cout << "Задайте месяц: ";
   cin >> m;           //  надо проверять
   cout << "Задайте день : ";
   cin >> day;         //  надо проверять
   for (int i = 0, s = 0; i < m – 1; ++i)
   s += month[i];      //  считаем месяцы
   s += day;           //  добавляем дни
   int weekDay = ((s + 2) % 7) ? (s + 2) % 7 : 7;     //  получаем день
                                                      //  недели
   cout << weekDay << endl;
   return 0;
}
```

Правильность задаваемых данных не проверяется, чтобы не загромождать программу. Почему написано условное выражение для вычисления пере-

менной weekDay? Мы привыкли, что воскресенье — седьмой день недели[1]. Для воскресений остаток от деления на 7 равен 0, а нам требуется число 7. Поэтому приходится различать эти случаи: воскресенье и все остальные дни. Проверка нашей программы показывает, что при условии задания корректных данных она работает правильно.

Обработка числовых массивов

Теперь давайте напишем несколько программ, с помощью которых продемонстрируем типичные варианты обработки массивов чисел:

- получение различных частичных сумм (в т. ч. и суммы всего массива);
- поиск максимума, минимума и других задаваемых значений;
- сортировка массива;
- выполнение однотипных действий над всеми элементами массива.

Реализуем эти задачи в виде функций, параметром которых будет массив. Сначала напишем функцию, вычисляющую сумму массива вещественных чисел. Нам, как обычно, требуется написать универсальную функцию, поэтому она должна работать для массивов любой длины. Заголовок такой функции выглядит так:

```
double Summa(double array[], int n)
```

По правилам языка C++ (которые в этом случае унаследованы от C), функция, получающая массив в качестве параметра, "не знает", сколько элементов в этом массиве. Поэтому мы и написали второй параметр — количество элементов массива. Попробуем, однако, вычислить количество элементов массива, используя для этого операцию sizeof, аналогично тому, как мы вычисляли количество элементов массива month. Тогда наша функция выглядит следующим образом:

```
double Summa(double array[])
{  unsigned long n = sizeof(array) / sizeof(double);
   double s = 0;
   for (int i = 0; i < n; ++i) s += array[i];
   return s;
}
```

Далее напишем main и вызовем нашу функцию:

```
int main()
{  double v[10] = { 1, 2, 3, 4, 5, 6 };
```

[1] Во времена Христа воскресенье было первым днем недели, а суббота — седьмым.

```
   cout << Summa(v) << endl;
   return 0;
}
```

Однако результат получается неожиданный — ноль! Такой результат может получиться только в двух случаях: либо массив содержит только нулевые значения, либо цикл в функции не выполняется ни разу, и функция возвращает первоначальное значение переменной `s`, равное нулю. Первый вариант сразу отпадает, второй вариант возможен, если переменная `n` — количество элементов массива — равна нулю. Для исследования работы программы и функции добавим операторы `cout`, которые покажут нам длину массива в основной программе и в функции.

```
double Summa(double array[])
{  cout << "Функция: " << sizeof(array) << endl;
   unsigned long n = sizeof(array) / sizeof(double);
   double s = 0;
   for (int i = 0; i < n; ++i) s += array[i];
   return s;
}
int main()
{  double v[10] = { 1, 2, 3, 4, 5, 6 };
   cout << "Программа: " << sizeof(v) << endl;
   cout << Summa(v) << endl;
   return 0;
}
```

Программа, как и положено, выводит число `80` (`sizeof(double) * 10`). А вот функция выводит число `4`. Теперь понятно, почему получается ноль в результате: целое деление `4 / sizeof(double)` равно `0`, цикл не выполняется ни разу, и функция возвращает ноль.

Однако это может означать только одно: сам массив в функцию не попадает. Таким образом, массив в функцию передается не по значению. Так как массивы появились еще в C, а ссылок тогда не было, то передача по ссылке тоже отпадает. Остается только один вариант из приведенных в *гл. 2* — по указателю. С указателями мы разберемся позже, а сейчас исправим нашу функцию, чтобы она вычисляла правильное значение. Текст новой функции приведен в листинге 3.2.

Листинг 3.2. Сумма массива

```
double Summa(double array[], int n)
{  double s = 0;
   for (int i = 0; i < n; ++i) s += array[i];
   return s;
}
```

Проверка показывает, что функция работает совершенно правильно, если верно задано количество элементов.

Однако функция получилась более универсальной, чем изначально задумывалось: с помощью этой функции можно вычислять "хвостовые" суммы, начиная с любого элемента массива. Нам надо только правильно задать конструкцию, с какого элемента начинать считать. Нельзя написать обычное выражение с индексом в скобках, например, `v[5]` — это значение пятого элемента массива, а не обозначение "хвоста" массива, начиная с 5-го элемента. В C++ это можно сделать, прибавив номер элемента к имени массива.

```
int main()
{  double v[10] = { 1, 2, 3, 4, 5, 6 };
   cout << Summa(v, 10) << endl;       //  весь массив
   cout << Summa(v + 2, 8) << endl;    //  последние 8 элементов
   cout << Summa(v + 5, 5) << endl;    //  5 последних элементов
   return 0;
}
```

Вместо номера элемента-константы мы даже можем задавать произвольное выражение, например:

```
int main()
{  int i = 3;
   const int ten = 10;
   double v[ten] = { 1, 2, 3, 4, 5, 6 };
   cout << Summa(v + i, ten – i) << endl;    //  v + i правильно
                                             //  вычисляется
   return 0;
}
```

Главное, чтобы значение этого выражения не было отрицательным, или больше количества элементов массива — тогда последствия непредсказуемы. Так как C++ не может проверить правильность этого выражения, то вся ответственность ложится на программиста. Например, никаких сообщений ни при трансляции, ни при выполнении не выдается, если программист напишет такой вызов:

```
cout << Summa(v, 10000) << endl;    //  ОШИБКА
```

Однако совершенно очевидно, что это серьезная ошибка, которую программисты обычно называют "вылет за границу". При работе с массивами программист совершенно не защищен от подобного рода ошибок.

Аналогично пишутся и другие функции, например поиск заданного значения в массиве. Функция (ее текст приведен в листинге 3.3) должна возвра-

щать номер найденного элемента. Если элемента в массиве нет, то возвращается -1.

Листинг 3.3. Поиск элемента в массиве

```
int FindNumber(double Number, double array[], int n)
{  for (int i = 0; i < n; ++i) if (array[i] == Number) return i;
   return -1;
}
```

Функция также прекрасно работает, если параметры передаются правильные. Как и для функции суммирования массива (см. листинг 3.2), в этом случае мы тоже имеем возможность искать заданное число в "хвосте" массива, например:

```
double pi = 3.141592653;
int k = FindNumber(pi, v + 3, n - 3);
```

Вообще-то существует один вариант передачи массива в качестве параметра, когда можно не указывать количество элементов: необходимо в качестве параметра объявить ссылку на массив. Текст функции суммирования, принимающей такой параметр, приведен в листинге 3.4.

Листинг 3.4. Параметр-ссылка на массив

```
const int n = 10;
double Summa(double (&array)[n])
{  double s = 0;
   for (int i = 0; i < n; ++i) s += array[i];
   return s;
}
```

Скобки вокруг имени параметра-массива указывать обязательно, без них программа не транслируется. К сожалению, такой способ не позволяет написать универсальную подпрограмму для обработки массива произвольного размера. В этом случае тип и количество элементов массива фиксированы, они должны указываться при объявлении. Поэтому в функцию при вызове допускается передавать массивы только указанного размера. Мы объявили глобальную константу и использовали ее в объявлении параметра-массива. При вызове такой функции мы уже не можем посчитать "хвостовые" суммы, т. к. со ссылками арифметические вычисления запрещены — программа просто не транслируется.

Оператор `typedef` позволяет прояснить ситуацию с передачей ссылки. Для массивов этот оператор имеет нестандартный синтаксис:

```
typedef тип Имя[количество];
```

Имя — это имя нового типа. Его можно использовать для объявлений массивов заданного типа и размера. Количество должно быть константным выражением. Таким образом, у нас в программе могут встречаться следующие объявления:

```
typedef double DoubleArray[10];
DoubleArray a;
```

Теперь заголовок функции Summa мы можем задать так:

```
double Summa(DoubleArray &a)    //  эквивалентно листингу 3.4
```

Для компилятора такой заголовок абсолютно идентичен прописанному нами ранее в листинге 3.4. В этом случае в теле функции мы обязаны использовать константу — количество элементов при организации цикла вычислений — переменную компилятор не пропустит. Попытки использовать это имя для возврата массива

```
DoubleArray Summa(DoubleArray &a)
```

так же не проходят — компилятор тут же обнаруживает "обман".

Если мы случайно или намеренно не укажем ссылку в параметре:

```
double Summa(DoubleArray a)
```

то нарвемся на очередной "риф" C++. Синтаксических ошибок в этом объявлении не наблюдается. Может показаться, что таким образом массив передается "по значению". Однако подобная запись является грубой ошибкой — в данном случае считается, что массив передается по указателю. А это означает, что надо передавать количество элементов, иначе результат вычислений непредсказуем.

И наконец, что произойдет, если у нас в программе объявлена и та и другая функция, т. е. мы перегружаем функцию Summa:

```
double Summa(DoubleArray &a) {//. . .};
double Summa(DoubleArray a) {//. . .};
```

Определения транслируются без ошибок. Однако при вызове функции возможны "аварийные ситуации" вплоть до отказа компилятора "выдать готовый продукт", т. е. работающую программу.

```
cout << Summa(v) << endl;         //  весь массив
cout << Summa(v + 2) << endl;     //  последние 8 элементов — опасный
                                  //  вызов
cout << Summa(v + 5) << endl;     //  5 последних элементов — опасный
                                  //  вызов
```

Первый оператор является двусмысленным: компилятор не может решить, какая функция вызывается. Остальные работают по варианту без ссылки,

т. к. ссылки не могут участвовать в арифметических выражениях. А из-за того, что в данном случае количество элементов не передается, то программа либо вообще не будет работать, либо посчитает неверную сумму.

Так как для копирования одного массива в другой требуется цикл, будет полезным реализовать эту операцию в виде функции. Более того, мы напишем шаблон, текст которого приведен в листинге 3.5, чтобы можно было копировать массивы любого числового типа.

Листинг 3.5. Копирование массивов

```
template < typename T >
void Copy(const T a[], T b[], int n)
{ for (int i = 0; i < n; ++i) b[i] = a[i]; }
```

Примечание

Чтобы этот пример заработал в системе Borland C++ 3.1, необходимо ключевое слово `typename` заменить словом `class`.

Мы прописали в списке параметров два массива — один массив описан как константный, а второй — нет. Дело в том, что у нас нет возможности возвратить массив как результат функции — если массив не попадает в функцию, то и возвратить массив функция не может. Поэтому нам нужен один входной массив и один выходной. Задокументируем в прототипе входной массив как константный, а выходной — нет. Без `const` по прототипу трудно сообразить, что куда пересылается. Проверяем наш шаблон для массивов двух типов: `int` и `double`.

```
int main()
{  int i;
   int v[10] = {0};
   int w[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   for (i = 0; i < 10; ++i) cout << v[i] <<' ';    //  выводит нули
   cout << endl;
   Copy(w, v, 10);    //   w -> v
   for (i = 0; i < 10; ++i) cout << v[i] <<' ';
   cout << endl;
   double v1[10] = {0};
   double w1[10] = { 1.1, 2.1, 3.1, 4, 5, 6, 7, 8, 9, 10.1 };
   for (i = 0; i < 10; ++i) cout << v1[i] <<' ';
   cout << endl;
   Copy(w1, v1, 10);    //  w1 -> v1
   for (i = 0; i < 10; ++i) cout << v1[i] <<' ';
   cout << endl;
   return 0;
}
```

Программа выводит на экран:

```
0 0 0 0 0 0 0 0 0 0              //  пустой массив v типа int
1 2 3 4 5 6 7 8 9 10             //  копия v <- w
0 0 0 0 0 0 0 0 0 0              //  пустой массив v1 типа double
1.1 2.1 3.1 4 5 6 7 8 9 10.1     //  копия v1 <- w1
```

Таким образом, если мы не прописываем слово `const`, то такой массив можно свободно модифицировать в теле функции. Этот пример наглядно демонстрирует, что передача массива по указателю очень похожа на передачу параметра по ссылке, но нам нет нужды указывать символ `&` при объявлении параметра-массива.

Теперь мы легко можем написать функцию, модифицирующую аргумент-массив некоторым заданным образом, например, умножает все элементы на 2 (ее текст приведен в листинге 3.6).

Листинг 3.6. Умножение массива на 2

```
template < typename T >
void Mult2(T array[], int n)
{ for (int i = 0; i < n; ++i) array[i] *= 2; }
```

Мы опять написали шаблон, чтобы иметь возможность оперировать с числовыми массивами любого типа.

При обработке массивов начинающий программист часто "наступает на одни и те же грабли". Обычно приходится выполнять некоторую операцию над всеми элементами массива, например, поделить на некоторое значение. Программист пишет функцию, в которой, для пущей универсальности, решает передавать это значение в качестве параметра: "делить — так делить".

```
void Divide(double array[], int n, double m)
{ for (int i = 0; i < n; ++i) array[i] /= m; }
```

Никакого "криминала" в определении не наблюдается. И в самом деле, любой из вызовов

```
double w[10] = { 1.1, 2.1, 3.1, 4, 5, 6, 7, 8, 9, 10.1 };
Divide(w, 10, 2);
Divide(w, 10, 3);
Divide(w, 10, 7.543);
```

проблем не вызывает — все работает именно так, как и задумывал программист. Где же ошибка? Вот она:

```
Divide(w, 10, w[5]);
```

Никаких сообщений об ошибках, естественно, не возникает. Однако результат работы функции совершенно не тот, на который рассчитывал про-

граммист: очевидно, требовалось поделить все элементы массива на пятый элемент. Сам пятый элемент должен получить значение `1`. Однако если мы выведем массив `w` на экран после вызова функции `Divide`, то увидим, что первые четыре элемента совершенно правильно поделены на пятый, и сам пятый элемент равен `1`. А вот после него все элементы остались без изменения. "Собака зарыта" именно в пятом элементе. После того как цикл выполнит деление для этого элемента, его значение станет равно `1`, и дальше производится деление на единицу!

Операция деления не является в данном случае особой — точно такая же ошибка возникает и при сложении, вычитании и умножении. Дело в аргументе, который является элементом массива. Подобного рода проблемы возникают, например, при программировании метода Гаусса для решения системы линейных уравнений — там нужно делить все элементы строки матрицы на главный элемент, который тоже содержится в матрице.

Индексы как параметры

Одним из распространенных приемов при работе с одномерными массивами является передача в качестве параметров начального и конечного индекса элементов. Особенно такой прием распространен в *рекурсивных функциях (см. гл. 7)*, однако и в простых циклических программах довольно часто применяется. Подобный способ обработки имеет дополнительные удобства: мы сможем обрабатывать любую последовательность элементов исходного массива, явно указывая индексы первого и последнего. Есть ряд традиционных задач обработки массивов, которые проще реализовать именно таким способом. Одной такой классической задачей является быстрая сортировка, второй — двоичный поиск в отсортированном по возрастанию массиве. Обеим задачам посвящено огромное количество работ, некоторые из которых приведены в [9, 14, 39, 41, 45], однако нас интересует в данном случае не теория, а чисто практический вопрос передачи параметров.

Реализуем функцию двоичного поиска в отсортированном массиве. Для определенности будем считать, что массив — целого типа. Как массив, так и искомое число, очевидно, должны передаваться функции как параметры. Функция должна вычислять индекс середины некоторого сегмента исходного массива. Поэтому в качестве параметров лучше передавать начальный и конечный индекс сегмента. Таким образом, заголовок функции выглядит следующим образом:

```
int BinarySearch(int a[], int begin, int end, int v)
```

где `begin` и `end` — начальный и конечный индекс элементов сегмента массива, а `v` — искомое число. Возвращать такая функция (ее текст приведен в листинге 3.7) будет номер искомого элемента, если он есть в массиве, или `-1`, если элемента в массиве нет.

Листинг 3.7. Двоичный поиск в массиве

```
int BinarySearch(const int a[], int begin, int end, int v)
{  while (begin <= end)
   {  int m = (begin + end) / 2;      //  середина последовательности
      if (v == a[m]) return m;
      if (v < a[m]) end  = m - 1;     //  "идем налево"
      else     begin = m + 1;         //  "идем направо"
   }
   return -1;                         //  в массиве нет элемента
}
```

Поскольку данная функция не изменяет элементы массива, то мы явно указываем этот факт, прописывая в заголовке слово `const`. Попробуйте написать двоичный поиск с параметром-количеством элементов — все равно придется перейти к индексам. Например, так:

```
begin = 0;
end = begin + (n - 1);
```

Теперь нам осталось только написать сортировку массива по возрастанию. Опять не будем вдаваться в теорию, а просто напишем шаблон для сортировки любых числовых массивов. Используем один из простых методов — сортировку выбором.

Идея этого метода заключается в том, что в массиве выбирается минимальный элемент, потом из оставшихся — опять минимальный и т. д. Можно помещать очередной выбранный элемент исходного массива на соответствующее место в массив-результат. Однако сортировки разрабатывались в те далекие времена, когда памяти у компьютеров было очень мало, использовать два массива просто не было физической возможности. Поэтому сортировки традиционно пишутся так, чтобы сортировать массив "на месте", не используя дополнительных массивов — очередной выбранный минимальный элемент должен переставляться с элементом того же массива. Первый минимум переставляется с первым элементом, затем из оставшихся опять выбирается минимальный и переставляется со вторым и т. д.

Результата-числа у функции нет — это "процедура". Текст шаблона сортировки массива приведен в листинге 3.8.

Листинг 3.8. Сортировка массива методом выбора

```
template < typename T>
void SortArray(T a[], int n)
{  for (int i = 0; i < n - 1; ++i)
   {  T min = a[i];
```

```
      for (int j = i + 1, k = i; j < n; ++j)
      if (a[j] < min) { min = a[j]; k = j; }
      a[k] = a[i];     //  обмен
      a[i] = min;      //  элементов
   }
}
```

При обмене используется переменная `min`, в которой уже записан `k`-й элемент массива, являющийся минимальным значением.

Проверка шаблона

```
double w1[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 10.1 };
for (i = 0; i < 10; ++i) cout << w1[i] << ' ';
cout << endl;
SortArray(w1, 10);
for (i = 0; i < 10; ++i) cout << w1[i] << ' ';
cout << endl;
```

показывает:

```
9 8 7 6 5 4 3 2 1 10.1    //  до сортировки
1 2 3 4 5 6 7 8 9 10.1    //  после сортировки
```

что она работает совершенно правильно. Так же, как и во всех предыдущих функциях, мы имеем возможность сортировать только "хвост" массива.

Напишем второй вариант той же сортировки, но параметрами будут индексы первого и последнего элемента массива. Вместо индекса последнего элемента будем задавать номер на 1 больше — так оказывается удобнее проверять условие окончания. Кроме того, такой вариант позволяет нам осуществить проверку правильности задания границ передаваемого массива: первый индекс должен быть меньше второго, по крайней мере, на 1, а сами индексы не могут быть меньше нуля. К сожалению, и в таком варианте мы не можем гарантировать отсутствие "вылета" за границы массива. Однако возможность хоть какой-то проверки параметров говорит в пользу такого метода. Поэтому, если параметры неверные, функция будет возвращать `-1`, а если все задано верно, то ноль. Текст шаблона сортировки массива с использованием параметров-индексов приведен в листинге 3.9.

Листинг 3.9. Сортировка с параметрами-индексами

```
template < typename T>
int SortArray(T a[], int first, int last)
{  if ((last <= first) || (first < 0) || (last <= 0)) return -1;
   for (int i = first; i < last – 1; ++i)
   {  T min = a[i];
```

```
      for (int j = i + 1, k = i; j < last; ++j)
      if (a[j] < min) { min = a[j]; k = j; }
      a[k] = a[i];
      a[i] = min;
 }
   return 0;
}
```

Как видите, мы просто перегрузили шаблон. Проверка его на массиве целого типа

```
int w[10] = { 10, 2, 3, 9, 8, 6, 5, 7, 4, 1};
for (i = 0; i < 10; ++i) cout << w[i] <<' ';
cout << endl;
SortArray(w, 0, 10);
for (i = 0; i < 10; ++i) cout << w[i] <<' ';
cout << endl;
```

показывает:

```
10 2 3 9 8 6 5 7 4 1    //  до сортировки
1 2 3 4 5 6 7 8 9 10    //  после сортировки
```

что все работает совершенно правильно. В таком варианте мы получаем возможность сортировать любую последовательность элементов в массиве.

Многомерные массивы

Многомерный массив в C++ рассматривается как одномерный, элементами которого являются массивы. Ограничений на количество измерений не накладывается — все зависит от реализации. Так как массив — это набор однотипных элементов, массивы-элементы многомерного массива все должны быть одного типа и размера. Опять на помощь приходит оператор `typedef`:

```
typedef float array[10];    //  array – массив из 10 float
array m[10];                //  массив из 10 массивов float[10]
```

Однако обычно двумерный массив объявляется более традиционным способом:

```
int a[10][10];
```

Как и другие переменные, многомерный массив можно инициализировать:

```
int A[3][3] = { { 1, 2, 3 }, { 2, 3 } };
```

Мы объявили массив из 3-х элементов, каждый элемент является массивом из 3 целых. Элемент `A[0]` получил значение `{ 1, 2, 3 }`, элемент `A[1]` равен `{ 2, 3, 0 }`, остальные элементы все равны `0`.

Элементы одномерного массива располагаются в памяти последовательно: `A[0]`, `A[1]`, `A[2]`. Это значит, что двумерный массив расположен в памяти по строкам. На рис. 3.1 показано, как размещается массив `A`.

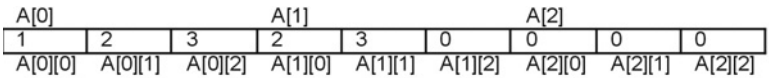

Рис. 3.1. Размещение в памяти массива `A[3][3]`

Доступ к отдельному элементу двумерного массива записывается как `A[i][j]`, где первый индекс — это номер строки, а второй — номер столбца.

Инициализацию многомерного массива можно выполнить и "одномерным способом:

```
int A[3][3] = { 1, 2, 3, 2, 3 };
```

Это объявление эквивалентно предыдущему. Как обычно, можно не указывать количество элементов, но только в самых левых скобках, например:

```
int A[][3] = { { 1, 2, 3 }, { 2, 3 } };
```

В данном случае считается, что массив `A` состоит из двух элементов (рис. 3.2), каждый из которых является массивом из 3 целых.

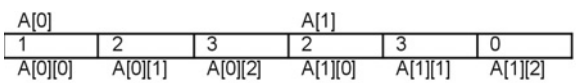

Рис. 3.2. Размещение в памяти массива `A[ ][3]`

При инициализации `A[0]` заполняется полностью, а в `A[1]` заносится только два числа, а третий элемент обнуляется.

Как многомерные массивы-параметры передаются в функции, рассмотрим в *гл. 4*.

Строки

Вернемся к программе, вычисляющей день недели (см. листинг 3.1). Давайте усовершенствуем ее — вместо вывода номера дня недели будем выводить соответствующее название. Названия месяцев поместим в массив. Тогда переменная `weekDay` может служить в качестве индекса, и вывод соответствующего названия дня недели на экран будет совсем простым:

```
cout << ДеньНедели[weekDay] << endl;
```

Но для этого мы должны решить проблему объявления массива строк — как известно, встроенного типа строк в C++ нет. Поэтому до поры до времени будем пользоваться массивами символов, как и было определено с самого начала в C — это работает в любой интегрированной среде. Как утверждают авторы языка C [17], самый распространенный вид массива — массив символов. До появления в стандарте C++ строкового типа данных `string` это же было справедливо и для C++.

Примечание

Тип `string` не является встроенным — он реализован как класс и входит в стандартную библиотеку шаблонов, которая не поддерживается средой Borland C++ 3.1.

Чтобы объявить одну строку, мы должны объявить массив типа `char`. Таким образом, чтобы объявить массив из 7 строк, необходимо объявить массив массивов или двумерный массив типа `char`. Начнем с объявления одной строки:

```
char s[] = "Понедельник";
```

Такое объявление не вызывает протестов компилятора, хотя по правилам инициализации массивов мы должны писать гораздо длиннее, а именно:

```
char s[] = { 'П', 'о', 'н', 'е', 'д', 'е', 'л', 'ь', 'н', 'и', 'к' };
```

Но такая запись, конечно, вызывает "протест" программистов, поэтому для массивов типа `char` (и только для них) разрешается "сокращенная" запись инициализации.

Так как у нас должен быть массив из 7 строк, необходимо задать максимальную длину одной строки. Посчитав количество символов в самом длинном слове, мы можем написать объявление массива с названиями дней недели:

```
char Days[][11] = { "Воскресенье", "Понедельник", "Вторник", "Среда",
                    "Четверг", "Пятница", "Суббота"
                  };
```

Заодно вернемся к древней нумерации дней недели, и избавимся от сложного выражения при вычислении остатка от деления. Текст обновленной программы для вычисления дня недели приведен в листинге 3.10.

Листинг 3.10. Вычисление дня недели

```
#include <iostream.h>
int main()
{  unsigned int day, m;
   int month[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

```
   char Days[][11] = {"Воскресенье", "Понедельник", "Вторник", "Среда",
                      "Четверг", "Пятница", "Суббота"
                     };
   begin:
   cout << "Задайте месяц (1-12): ";
   cin >> m;
   if ((1 > m) || (m > 12)) goto begin;           //  если неверно
   mm: cout << "Задайте день : ";
   cin >> day;
   if ((1 > day) || (day > month[m])) goto mm;    //  если неверно
   for (int i = 0, s = 0; i < m - 1; ++i)
   s += month[i];                //  считаем месяцы
   s += day;                     //  добавляем дни
   int weekDay = (s + 2) % 7;    //  получаем день недели
   cout << Days[weekDay] << endl;
   return 0;
}
```

При выполнении программы в среде Borland C++ 3.1 для воскресенья выводится подряд три названия:

```
ВоскресеньеПонедельникВторник
```

а для понедельника — два:

```
ПонедельникВторник
```

Дело в том, что по принятой еще в C идеологии константы-строки должны заканчиваться байтом-нулем, который является признаком окончания строки. Этот байт-нуль автоматически добавляется в конце каждой константы, увеличивая ее размер на один байт. Убедиться в этом можно, выполнив простую программу:

```
#include <iostream.h>
int main()
{  cout << sizeof("Понедельник")  << endl;
   cout << sizeof("Вторник") << endl;
   return 0;
}
```

На экране появится число `12` и число `8` — это показывает, что в памяти данные константы-строки занимают на 1 байт больше, чем количество видимых символов. Таким образом, в памяти компьютера слово "понедельник" и слово "воскресенье" должны занимать 12 байт, а не одиннадцать, как мы задали в массиве. На байт-нуль при нашем объявлении для "воскресенья" и "понедельника" места не выделено, поэтому вывод выполняется до ближайшего нуля, а это — нуль в конце "вторника".

Примечание

В системе Visual C++ 6 данная программа даже не транслируется, а системы фирмы Borland "обманывают" программиста, пропуская такую ошибку.

Таким образом, мы должны объявить массив дней недели, задав количество символов, равное 12.

```
char Days[][12] = {"Воскресенье", "Понедельник", "Вторник", "Среда",
                   "Четверг", "Пятница", "Суббота"
                  };
```

Тогда программа работает совершенно правильно. Таким образом, нам необходимо различать понятия *символьный массив* и *символьная строка*. Каждая символьная строка является символьным массивом, но не каждый символьный массив является символьной строкой — отличительным признаком является нулевой байт в конце символьной строки.

Обработка символьных массивов

Поскольку строки (массивы символов) заканчиваются нулями, при написании функций обработки строк нет необходимости задавать длину массива. Мы легко можем написать функцию (ее текст приведен в листинге 3.11), вычисляющую длину строки.

Листинг 3.11. Вычисление длины символьного массива

```
unsigned int Length(const char s[])     //  количество элементов — не
                                        //  нужно
{  int L = 0;
   while (s[L++]);     //  все вычисления — в условии цикла
   return (L — 1);
}
```

Функция не изменяет массив, поэтому мы задали константность. Цикл будет выполняться до тех пор, пока очередной символ строки `s` не равен нулю. Как только функция "натыкается" на нулевой байт, завершающий строку, цикл заканчивается. В этот момент в переменной `L` находится число, на 1 большее количества символов в строке — поэтому возвращается выражение `L — 1`.

Теперь разберемся с переводами чисел. Любая программа выполняет такой перевод при вводе (с клавиатуры вводятся символы) и обратный — при выводе (на экран тоже выводятся символы). Напишем функцию (ее текст приведен в листинге 3.12), аналогичную системной, которая из символьного массива получает целое число. Параметром такой функции будет массив символов, а результатом — целое число.

Листинг 3.12. Перевод числа atoi

```
int atoi (const char number[])
{  int n = 0,     //  получаемое число
   sign = 1;      //  знак числа
   int i = 0;
   if (number[i] == '+') ++i;
   if (number[i] == '-') { sign = -1; ++i; }
   while (i < Length(number))
   { n = n * 10 + (number[i] - '0'); ++i; }
   n *= sign;     //  "добавляем" знак
   return n;
}
```

Число накапливаем в переменной `n`. Разберем выражение

```
n = n * 10 + (number[i] - '0');
```

по частям. Выражение `(number[i] - '0')`, как мы знаем, дает нам значение цифры. Затем предыдущее значение `n` умножается на `10`, и к нему добавляется значение цифры. Пусть, например, в `number` находится строка `5472`. Первый раз в `n` находится `0`, поэтому после первого шага цикла `n` станет равно `5`. На втором шаге:

```
n = 5 * 10 + 4 = 54;
```

На третьем шаге получим:

```
n = 54 * 10 + 7 = 547;
```

И наконец, на последнем, четвертом шаге вычисляется окончательное значение:

```
n = 547 * 10 + 24 = 5472;
```

Переменная `sign` позволяет нам отследить знак минус. Вызов такой функции может быть следующим:

```
char s[] = "-5427";
int a = atoi(s);
int b = atoi("3456");
```

Следите за тем, чтобы аргумент не превосходил пределов целых чисел. Мы могли бы написать и другие аналогичные функции преобразования, однако функция `atoi`, а также `atol` и `atof`, есть в стандартной библиотеке stdlib.h. Тем не менее всегда полезно разобраться во внутренностях работы — это повышает квалификацию.

Несмотря на то, что константа-строка выглядит как скалярная величина, присваивать константу-строку символьному массиву — нельзя. Однако передавать в качестве фактического параметра — можно. Можно даже воз-

вращать в качестве результата. Но для присвоения строк надо писать функцию, аналогичную функции копирования числовых массивов (см. листинг 3.4). Количество элементов задавать нет необходимости — надо только следить за тем, чтобы длина принимающего массива была не меньше отдающего. Используем для этого только что написанную функцию `Length` (см. листинг 3.11). Функция копирования (ее текст приведен в листинге 3.13) возвращает `0`, если копирование выполнилось нормально, и `-1`, если принимающий массив короче исходного. Как и в программе копирования числовых массивов, входной массив объявлен константным, а выходной — нет.

Листинг 3.13. Копирование символьных массивов

```
int CopyStr(const char source[], char dest[])
{  if (Length(source) < Length(dest)) return -1;
   for (int i = 0; i < Length(source); ++i) dest[i] = source[i];
   return 0;
}
```

Вызов такой функции может быть таким:

```
char s[80] = {0};    //  "пустая" строка
int cc = CopyStr("Копируемая строка", s);
```

или таким:

```
char ss[80] = "";    //  "пустая" строка
char dd[] = "Исходная строка";
int cc = CopyStr(dd, ss);
```

Мы в этом примере продемонстрировали два разных способа "обнуления" строки. И тот и другой способы полностью обнуляют массив, в чем можно убедиться, выполнив посимвольный вывод массива на экран:

```
for (i = 0; i < 80; ++i) cout << (int) s[i] << ' ';
cout << endl;
for (i = 0; i < 80; ++i) cout << (int) ss[i] << ' ';
cout << endl;
```

Мы задали преобразование в целое число, т. к. код, равный нулю, на экран выводится как пробел.

Ввод и вывод массивов символов

Наконец, разберемся с вводом/выводом строк. В программе вывода дня недели (см. листинг 3.10) массив строк — двумерный, а мы выводим один элемент двумерного массива без всякого цикла:

```
cout << Days[weekDay] << endl;
```

Язык C++ и здесь следует за C, разрешая выводить на экран без использования оператора цикла символьные массивы-строки. Достаточно написать имя массива в операторе вывода, например:

```
char s[] = "Понедельник";
cout << s << endl;
```

Как мы знаем, при инициализации того же массива посимвольно

```
char s[] = { 'П', 'о', 'н', 'е', 'д', 'е', 'л', 'ь', 'н', 'и', 'к' };
```

никакого дополнительного байта не выделяется. В этом легко убедиться, вычислив размер массива с помощью операции `sizeof`. Следующий оператор выведет на экран число `11` — по количеству элементов-символов:

```
cout << sizeof(s) << endl;
```

Поскольку в конце нет нуля, то с выводом такого массива могут возникнуть проблемы. Нижеприведенная программа

```
#include <iostream.h>
int main()
{  char b = 'a';
   char s[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
   int d = 10101;
   cout << sizeof(s) << endl;
   cout << s << endl;
}
```

в среде Borland C++ 3.1 выводит на экран следующее:

```
6
abcdefwa
```

Число `6` показывает размер массива `s`. А ниже — наш массив символов, но с "довеском" `wa`. Выполнив тот же пример в системе Visual C++ 6, получаем еще более странный "довесок":

```
6
abcdef¦¦a¦¦¦8_e
```

Совершенно очевидно, что к таким странным результатам приводит отсутствие завершающего нуля.

Необходимо совершенно ясно понимать, что размер массива и длина строки символов, которая в этом массиве хранится — это разные вещи. Для определения количества "значащих" символов строки в массиве используем определенную нами функцию `Length` (см. листинг 3.11). Эта функция по своему действию отличается от операции `sizeof`, что наглядно демонстрирует следующий фрагмент программы:

```
char s[10] = "Привет!";
cout << sizeof(s) << endl;     //  выведет 10
cout << Length(s) << endl;     //  выведет 7
```

Как мы видим, первый оператор `cout` выводит размер массива, а второй — количество символов в константе-строке.

Ввод символьных массивов-строк также отличается от ввода "обыкновенных" массивов. Так же, как при выводе, можно не использовать цикл, а просто задать в операторе ввода имя массива:

```
cin >> s;
```

Массив, естественно, должен быть объявлен с указанием количества элементов:

```
char s[80];
```

Однако и тут нас ожидают сюрпризы. Выполним следующую программу:

```
#include <iostream.h>
int main()
{  char sd[80];
   cin >> sd;
   cout << sd << endl;
   return 0;
}
```

Зададим строку из нескольких слов "привет от строки" — ввод заканчиваем клавишей <Enter>. Никаких проблем вроде бы не возникло. Однако оператор вывода покажет на экране только первое слово "привет". Такая ситуация наблюдается при выполнении программы во всех интегрированных средах. Попытка выполнить вывод строки в цикле посимвольно

```
for (int i = 0; i < 80; ++i) cout << sd[i];
```

показывает, что в массиве действительно находится только слово "привет". Таким образом, операция `>>` для символьных массивов работает только до пробела. Как вводить строки с пробелами, мы разберемся в *гл. 6*, где рассмотрим систему ввода/вывода C++.

Структуры

Структуры — это объединение в одну группу нескольких разнородных (или однородных) данных, каждое из которых имеет собственное имя. Например, можно объединить в структуры отдельные элементы даты: год, месяц и день. На языке C++ это выглядит так:

```
struct Date { unsigned int year, month, day; };
```

или так, как приведено в листинге 3.14.

Листинг 3.14. Объявление структуры

```
enum Month
{ jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
struct Date
{  unsigned int year;
   Month month;    //  использовали перечислимый тип
   unsigned int day;
};
```

Примечание

При объявлении типа `Month` с такими именами констант в системе Visual C++ 6 программист рискует "нарваться" на самую кошмарную ошибку, которую только можно представить: `fatal error C1001: INTERNAL COMPILER ERROR`, — очевидно понятную и без перевода. Проблема в том, что в объявлении типа `Month` мы написали два имени, которые уже прописаны в стандартной библиотеке шаблонов — `oct` и `dec`. Если мы переобъявим наш тип `Month`, задав имена с большой буквы или дописав в конце имен знак подчеркивания (`Oct` или `oct_`), то эта "страшная" ошибка исчезнет. Поэтому возьмите себе за правило объявлять свои имена либо с большими буквами, либо с подчеркиванием в конце.

Б. Страуструп постановил, что "структура является классом", поэтому слово `Date`, которое называется *тегом структуры*, вводит новое имя типа. После такого определения разрешается объявлять и инициализировать переменные типа `Date` несколькими разными способами, например:

```
Date today;                    //  без инициализации
Date d1 = { 2003, feb, 22 };   //  поэлементная инициализация
Date d = { 2000, jan };        //  неполная инициализация
Date d2 = d1;                  //  инициализация другой структурой
Date d3(d2);                   //  инициализация другой структурой
```

Переменные внутри структуры часто называют *полями*. Имена полей, естественно, должны отличаться друг от друга и от имени тега структуры. В C++ не разрешается инициализировать отдельные поля внутри структуры, можно инициализировать только всю структуру целиком. Допускается инициализация и с явным указанием полей:

```
Date z = { z.year = 2004, z.month = feb, z.day = 28 };
```

Никаких сообщений и предупреждений не выводится, однако имена полей в этом случае не играют никакой роли — инициализация происходит по порядку следования значений.

Например, при таких объявлениях

```
struct Old { int a; float b; char ch; };
Old z = { z.b = 2.5, z.ch='0' };
```

поле `a` получит значение `2` (от значения `2.5` отсекается `0.5`), в поле `b` окажется число `48` (код символа `'0'`), а поле `ch` обнулится.

Можно определение типа, объявление переменной и инициализацию совместить, например:

```
struct { unsigned int year, month, day; } d1 = { 2003, feb, 22 };
```

Однако при таком объявлении отсутствует возможность объявлять переменные типа `Date`, поэтому объявление отдельно структуры, отдельно переменных более удобно.

Для доступа к отдельному элементу структуры используется операция "точка" — *селектор* поля структуры. Например, чтобы использовать в программе отдельные поля переменной `d1`, мы должны писать `d1.year`, `d1.month`, `d1.day`. Однако, в отличие от массивов, структуры одного типа можно присваивать друг другу:

```
d1 = today;
```

Тем не менее ввод данных в структуру выполняется поэлементно, например:

```
cout << "Задайте дату (год месяц день):";
cin >> d1.year >> d1.month >> d1.day;
```

Вывод также осуществляется поэлементно:

```
cout << d1.year << '-' << d1.month << '-' << d1.day;
```

Структуры как параметры

Поскольку в C++ структура является классом по умолчанию, то при объявлении структуры мы объявляем новый тип объектов, равноправный со встроенными типами. Поэтому структуру можно не только передавать в качестве параметра, но и возвращать в качестве результата. При этом допускаются все возможные способы передачи параметров. Текст примера передачи структуры по значению приведен в листинге 3.15.

Листинг 3.15. Передача структуры по значению

```
struct Old { int a; float b; };
Old g(Old p)                //  функция получает и возвращает структуру
```

```
{  p.a = 6;
   p.b = 7;                //  изменения только внутри функции
   return p;               //  возврат структуры в качестве значения
}
int main(void)             //  главная функция
{  Old y = { 1, 2 }, x;    //  объявление переменных
   x = g(y);               //  переменная y не меняется
   y = g(y);               //  изменение переменной y при присваивании
}
```

В этом примере объявляется структура типа `Old` и определяется функция, принимающая параметр типа `Old` по значению и возвращающая результат того же типа. Вызов `x = g(y)` присваивает переменной `x` значения `x.a = 6` и `x.b = 7`; значение параметра `y` при этом не меняется, поскольку выполнялась передача по значению. Во втором случае переменная `y` тоже передается по значению, но возвращаемый из функции результат изменяет значения ее компонентов.

Передача по ссылке для структур совершенно аналогична передаче встроенных типов, т. е. передаваемая в качестве фактического параметра структура изменяется. Текст примера передачи структуры по ссылке приведен в листинге 3.16.

Листинг 3.16. Передача структуры по ссылке

```
struct Old { int a; float b;};
Old g(Old &p)    //  параметр — ссылка на структуру
{  p.a = p.b + 6;
   p.b = p.a + 7;
   return p;
}
int main(void)
{  Old y = { 1, 2 }, x;
   x = g(y);     //  структура y тоже изменяется
}
```

То есть вызов `x = g(y)` приводит к тому, что значения переменных `x` и `y` становятся одинаковыми.

Использование слова `const` при передаче параметров по ссылке приводит к тому, что в теле функции синтаксически запрещены присваивания (и ввод) такому параметру. В приведенном примере, текст которого представлен в листинге 3.17, это сделало необходимым объявить внутреннюю локальную переменную `r`.

Листинг 3.17. Передача структуры по константной ссылке

```
struct Old { int a; float b;};
Old g(const Old &p)     //  внутри функции параметру присваивать нельзя
{  Old r;               //  поэтому объявляется локальная переменная
   r.a = p.b + 6;
   r.b = p.a + 7;
   return r;
}
int main(void)
{ Old y = { 1, 2 }, x;
 x = g(y);               //  переменная y не изменяется
 y = g(y);               //  переменная y получила новое значение
}
```

Как и в случае других типов параметров, параметру-структуре разрешается присваивать значение по умолчанию. Однако в случае с параметром-структурой синтаксис инициализации использовать не разрешается:

```
Old g(Old p = { 1, 2 })
```

Такое присвоение начального значения параметру-структуре не проходит. Однако допускается иной стандартный вид инициализации — другой структурой, которая определена ранее:

```
Old g(Old p = t)
```

Однако второй вариант инициализации структурой

```
Old g(Old p(t))
```

считается ошибочным, если в структуре не определены *конструкторы*.

Самым интересным способом присвоения начального значения является вызов функции. Естественно, такая функция должна возвращать в качестве результата структуру такого же типа. Параметры у нее могут отсутствовать, но чаще в качестве параметра используется структура того же типа. Как и в случае простых параметров, допускается в качестве инициализирующей функции вызвать саму определяемую функцию, прописав прототип (его текст приведен в листинге 3.18).

Листинг 3.18. Присвоение значения по умолчанию параметру-структуре

```
struct Old              //  объявление структуры
{ int a; float b; };
Old y = { 1, 2 }, x;    //  объявление переменных
Old g(Old p);           //  прототип
Old g(Old p = g(y))     //  функция получает и возвращает структуру
```

```
{  p.a = 6;
   p.b = 7;              //  изменения только внутри функции
   return p;             //  возврат структуры в качестве значения
}
void main(void)
{  x = g();              //  используется значение по умолчанию
   cout << "y =" << y.a << ";" << y.b << endl;
   cout << "x =" << x.a << ";" << x.b << endl;
   y = g(y);             //  изменение переменной y
   cout << "y =" << y.a << ";" << y.b << endl;
}
```

В приведенной программе для присвоения значения по умолчанию используется тот же прием вызова самой определяемой функции. Поэтому выше указывается прототип (без него возникает ошибка трансляции). Программа выводит на экран следующее:

```
y = 1;2
x = 6;7
y = 6;7
```

Шаблоны структур

Очень часто используются структуры, в которых определены два поля, часто одного типа. Это могут быть комплексные или рациональные числа, например

```
struct Complex { double re, im; };
```

или

```
struct Rational { unsigned long num, denum; };
```

Это могут быть координаты точек на экране или на плоскости:

```
struct Point { double x, y; };
```

Учитывая "массовость явления", хотелось бы иметь вместо всех этих объявлений один шаблон. Для структур разрешено объявлять *шаблоны*. Все наши "пары" в шаблонном виде выглядят так:

```
template <class T>
struct Pair { T first; T second; };
```

Мы можем объявить разные типы для полей структуры-шаблона, текст которого приведен в листинге 3.19.

Листинг 3.19. Объявление шаблона-пары

```
template <class T1, class T2>
struct Pair
{ T1 first; T2 second; };
```

Это определение уже очень похоже на то, что реализовано в стандартной библиотеке шаблонов.

В отличие от структур, шаблон структуры может быть объявлен только вне любой функции, в т. ч. и вне функции `main` — иначе компилятор просто отказывается транслировать программу. А вот объявлять переменные с помощью такого шаблона можно как локально, так и глобально. Форма объявления, конечно, отличается от обычной, поскольку при объявлении мы должны указать конкретные типы вместо параметров шаблона. Это делается так:

```
Pair <double, double> a;              //  без инициализации
Pair <double, double> t = { 1, 2 };   //  инициализация полей
Pair <double, double> r = t;          //  инициализация структурой
Pair <double, double> d(t);           //  инициализация структурой
Pair <int, double> p = { 1.1, 2 };
```

В последнем случае Visual C++ 6 выдает предупреждение о возможной потере информации при присваивании дробного числа целому полю.

При использовании шаблонов более наглядно видно, что структуры разных типов не разрешается присваивать друг другу:

```
Pair <double, double> t = { 1, 2 };
Pair <int, double> k = t;
```

В этом случае компилятор, естественно, выдает сообщение об ошибке.

В качестве типа элемента пары можно указывать не только встроенные, но и определенные программистом, например, наш тип `Date`. Такая пара объявляется следующим образом:

```
Pair <Date, double> rr;
```

Для сокращения записи нам опять пригодится оператор `typedef`.

```
typedef Pair <double, double> ddPair;
typedef Pair <int, double> idPair;
```

При объявлении нового имени мы использовали обычный прием — обозначение типа с помощью префикса прямо в имени. Особенно широко этот прием применяется в Windows, где в системных включаемых файлах определено огромное количество типов.

Массивы и структуры

Массивы и структуры прекрасно уживаются друг с другом: можно объявлять массив внутри структуры и можно объявить массив структур, можно использовать уже определенную структуру для объявления поля новой структуры. Например, пусть требуется ежедневно обрабатывать набор из 10 измеряемых величин. Тогда нам может потребоваться следующая структура:

```
struct Number {double number[10]; Date date; };
```

Такую структуру можно инициализировать обычным образом, только надо в качестве первого инициализирующего элемента задать инициализатор массива:

```
Number N = { { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 0},
             { 2003, feb, 2 } };
```

Если измерения выполнялись в течение 100 дней, то в программе можно объявить массив:

```
Number NN[100];
```

Проинициализировать весь массив, конечно, сложно. Однако присвоение значений отдельным элементам массива вполне могут потребоваться в программе. Как обычно, такое присвоение выполняется поэлементно:

```
NN[0].number[0] = 1.2;
NN[0].number[1] = 3.4;
// . . .
NN[0].number[9] = 5.7;
NN[0].date.year = 2003;
NN[0].date.month = feb;
NN[0].date.day = 22;
```

Объявление массива внутри структуры позволяет делать нам то, что нельзя делать непосредственно с массивами — присваивать один массив другому. Пусть у нас объявлена структура:

```
struct Array { double a[10]; };
Array a = { {1, 2, 3, 4, 5, 6, 7, 8, 9,0 } };    //  инициализация
                                                 //  массива
Array b = a;
Array c;
c = a;
```

При этом можно даже не указывать внутренние фигурные скобки при инициализации элемента структуры:

```
Array a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };      //  это легально
```

Очень часто в реальных производственных программах требуется обрабатывать массивы структур, элементами которых являются массивы символов. Например, в отделе кадров хранятся карточки сотрудников, в которых указана разнообразнейшая информация, начиная с фамилий сотрудников и их дней рождения. Так как инициализировать поля внутри структуры нельзя, массив внутри структуры должен быть задан с явным количеством элементов. Пример объявления массива символов в структуре выглядит так:

```
struct Person { char fio[30]; Date BirthDay; };
```

Инициализировать такую одиночную структуру можно без проблем:

```
Person a = { "Страуструп", { 1955, 12, 21 } };
```

Инициализация массива структур тоже сложностей не представляет:

```
Person w[2] = { { "Страуструп", {1955, 12, 21 } },
                { "Эллис", { 1975, 10, 12 } }
              };
```

Для группировки инициализирующих элементов, связанных друг с другом, используются фигурные скобки.

Теперь объявим массив типа `Person` без инициализации:

```
Person mans[100];
```

Присвоить значения полю `BithDay` элемента массива `mans` можно как обычно:

```
mans[5].BirthDay.year = 2003;
mans[5].BirthDay.month = feb;
mans[5].BirthDay.day = 22;
```

Однако с присвоением значения фамилии возникают проблемы, поскольку массиву нельзя присвоить константу-строку:

```
mans[5].fio = "Фамилия";    //  не работает
```

Такое присвоение не проходит. А объявить массив неопределенной длины без указания количества элементов мы тоже не можем, т. к. такой массив должен быть проинициализирован. Эта проблема решается несколькими способами: использованием функций обработки строк из библиотеки string.h, заменой символьного массива полем типа `string`, реализованного в стандартной библиотеке шаблонов, или реализацией полноценного класса `Person` с перегрузкой операций и конструкторами.

Структуры, функции и шаблоны

Функции можно задавать как элементы структуры. Это прямое следствие декларирования структуры как класса. Опять рассмотрим пример с обработкой массива измеряемых чисел. Пусть обработка заключается в вычислении

среднего арифметического значения. При использовании функции суммирования (см. листинг 3.2) наша независимая функция выглядит так (ее текст приведен в листинге 3.20).

Листинг 3.20. Независимая функция вычисления среднего

```
double Average(double array[], int n)
{ return Summa(array, n) / n; }
```

Для вычисления среднего значения массива структуры `N` типа `Number` эту функцию вызывают так:

```
double a = Average(N.number, 10);
```

Однако нам не нужна универсальная функция для любого массива, нам нужна конкретная, которая "чисто конкретно" работает с массивом в нашей структуре. Поэтому, следуя принципу инкапсуляции, лучше объявить такую функцию элементом структуры. Часто ее называют *компонентной функцией*, Б. Страуструп называет такую функцию *функцией-членом*, однако мы будем придерживаться более традиционной объектно-ориентированной терминологии и назовем ее *методом*. Это название принято практически во всех объектно-ориентированных языках, в т. ч. — в Java.

Функцию можно прописывать в любом месте структуры, как до объявления полей, так и после. Пример объявления структуры с функцией приведен в листинге 3.21.

Листинг 3.21. Функция в структуре

```
struct Number
{  double number[10];
   Date date;
   double Average(void) { return Summa(number, 10) / 10; }
};
```

Если мы не объявим переменную типа `Number`, то и вызвать метод не сможем. Вызов метода выполняется таким же образом, как и доступ к полю структуры:

```
cout << N.Average() << endl;
```

Еще следует обратить внимание на то, что параметры этой функции методу не нужны. По умолчанию метод имеет доступ ко всем полям структуры.

Реализация метода у нас выполнена прямо в структуре, но это только потому, что он короткий. Реализованный непосредственно внутри структуры метод по умолчанию считается inline-функцией. В общем виде в структуре

прописывается только прототип, а реализация выполняется вне структуры. Но тогда надо как-то указать, что функция является методом, причем именно данной структуры, а не другой. Это делается с помощью операции разрешения контекста::.

```
double Number :: Average(void) { return Summa(number, 10) / 10; }
```

Здесь нас поджидает одна проблема: константа 10 используется и в структуре, и в методе. Если метод реализован отдельно от структуры, то при изменении количества элементов нам придется тщательно отследить все места программы, где мы используем эту константу. Пока программа небольшая, это не доставит неудобств, но в большой программе это может превратиться в проблему. Поэтому надо количество элементов массива задать с помощью именованной константы, но хотелось бы, следуя принципу инкапсуляции, объявить константу прямо в структуре. Непосредственно объявить константу как поле структуры мы не можем, поскольку ее требуется инициализировать, а это запрещено. Но у нас есть возможность объявить в структуре перечислимый тип enum, который, как известно, является набором поименованных целых констант. Пример объявления константы в структуре приведен в листинге 3.22.

Листинг 3.22. Константа в структуре

```
struct Number
{  enum { k = 10 };
   double number[k];
   Date date;
};
double Number :: Average(void) { return Summa(number, k) / k; }
```

Раньше (до принятия стандарта) использовались статические поля-переменные,

```
static int k;
```

которые надо было отдельно инициализировать в программе

```
int Number :: k = 10;
```

Однако такому способу присущи все описанные выше проблемы. Поэтому стандарт разрешил объявить статическую константу как поле структуры:

```
static const int k = 10;
```

Такая константа должна быть где-нибудь объявлена без инициализатора:

```
const int Number :: k;
```

Примечание

Стандарт разрешает такие объявления, однако ни Visual C++ 6, ни Borland C++ 5 не реализуют такую возможность. Подобная конструкция уже реализована в более новой версии системы Borland C++ Builder 6.

Решения приемлемые, следуют принципу инкапсуляции. Есть только одно но: все переменные типа `Number` будут содержать массив фиксированного размера в 10 элементов. Однако мы знаем, что в программировании нет ничего более постоянного, чем регулярные изменения, поэтому сегодня нам нужен массив из 10 элементов, а завтра — из 12, а послезавтра — только из 7. Не хотелось бы каждый раз исправлять исходный код и заново транслировать программу. И тут нам на помощь неожиданно приходят шаблоны. Такую нужную нам константу мы можем задать в качестве параметра шаблона, текст которого приведен в листинге 3.23.

Листинг 3.23. Шаблон структуры с целым параметром

```
template <int n>
struct Number
{  double number[n];
   Date date;
   double Average(void) { return Summa(number, n) / n; }
};
```

Мы можем реализовать метод и вне шаблона. Тогда шаблон вместе с "внешним" методом, текст которого приведен в листинге 3.24, должен быть объявлен таким образом:

Листинг 3.24. Шаблон структуры с "внешней" реализацией метода

```
template <int n>
struct Number
{  double number[n];
   Date date;
   double Average(void);
};
template < int n >
double Number <n> :: Average(void) { return Summa(number, n) / n; }
```

Обратите внимание, шаблон `template` прописывается и перед структурой, и перед реализацией метода. Доступ к полям и методам шаблона выполняется совершенно так же, как и к полям структуры, не являющейся шаблоном. Использовать шаблон структуры можно так:

```
Number<12>N = { { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 0 },
                { 2003, feb_, 2 } };
cout << N.Average() << endl;    //  среднее 12 чисел
Number<10>M = { { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 0 },
                { 2003, feb_, 2 } };
cout << M.Average() << endl;    //  среднее 10 чисел
```

В первом случае метод вычисляет среднее от 12 элементов массива (последние 2 равны нулю) и выдает `4.125`. Во втором варианте вычисляется среднее от 10 элементов массива и получается результат `4.95`. Кстати, если количество элементов в инициализирующем выражении будет больше, чем задано в константе, то возникает ошибка трансляции типа "слишком много инициализаторов".

В отличие от шаблонов функций, мы можем задавать значение константы по умолчанию. Только в таком случае выгоднее использовать шаблон с "внутренней" реализацией метода, текст которого приведен в листинге 3.25.

Листинг 3.25. Шаблон структуры с целым параметром

```
template <int n = 10>
struct Number
{  double number[n];
   Date date;
   double Average(void) { return Summa(number, n) / n; }
};
```

Такое определение позволяет нам как задавать другое значение параметра, так и не задавать его вовсе:

```
Number<12>N = { { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 0 },
                { 2003, feb_, 2 } };
cout << N.Average() << endl;    //  среднее 12 чисел
Number<>M = { { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 0 },
              { 2003, feb_, 2 } };
cout << M.Average() << endl;    //  среднее 10 чисел
```

Скобки пустые, т. к. мы уже специализировали шаблон при объявлении. В этом случае, как и для обычного массива, количество инициализаторов не должно превышать задаваемую константу.

Если просто структуры мы можем задавать как локально, так и глобально, то шаблон мы обязаны сделать глобальным. Однако это небольшая плата за универсальность. Естественно, структура `Date` должна быть прописана раньше.

Размеры структур

В связи с возможностью объявления в структуре функций-методов возникает вопрос о размерах таких структур. Исследуем эту проблему в Visual C++ 6. Текст программы определения размеров структур приведен в листинге 3.26.

Листинг 3.26. Размеры структур

```
#include <iostream.h>
int main()
{  struct AAA { char ch; double b; };
   struct BBB { double b; char ch; };
   struct CCC { int b; char ch; };
   struct DDD { double b; char ch;
                void f(void) { cout << "Hello!" << endl; }
                };
   cout << "AAA " << sizeof(AAA) << endl;
   cout << "BBB " << sizeof(BBB) << endl;
   cout << "CCC " << sizeof(CCC) << endl;
   cout << "DDD " << sizeof(DDD) << endl;
   return 0;
}
```

При выполнении данная программа выводит на экран:

```
AAA 16
BBB 16
CCC 8
DDD 16
```

Поскольку мы в настройках интегрированной среды и компилятора ничего не меняли, это означает, что в отладочном режиме по умолчанию будет следующее:

- независимо от размеров и размещения в структуре отдельных полей система выделяет количество байт, кратное наибольшему размеру. В нашем примере мы имеем для структуры `AAA` и `BBB` размер `16`, т. к. размер `double` равен `8`. А для структуры `CCC` выделено `8` байт памяти, т. к. `int` занимает `4` байта;
- наличие в структуре определения метода никак не увеличивает размеры структуры. Метод размещается системой в другом месте работающей программы.

Те же числа выдаются и в режиме трансляции **Release** с оптимизацией **Minimize Size**. Borland C++ Builder 6 и в отладочном, и в режиме **Release**

выдает те же числа. Та же программа в системе Borland C++ 3.1 выведет на экран совсем другие числа:

```
AAA 10
BBB 10
CCC 4
DDD 10
```

Добавляя в программу структуры с различными полями и разным порядком полей, можно сделать выводы, что в отладочном режиме по умолчанию будет следующее:

- независимо от количества и расположения полей в структуре Borland C++ 3.1 выделяет для каждого поля, кроме поля типа `char`, именно столько памяти, сколько требуется;
- для полей типа `char` выделяется `2` байта;
- наличие в структуре определения метода никак не увеличивает размеры структуры. Метод размещается системой в другом месте работающей программы.

Чтобы заставить компилятор выделять для элементов структур ровно столько памяти, сколько занимает соответствующий `sizeof(тип)`, нужно использовать директиву препроцессора `#pragma`. Хотя конструкция реализована как одна из директив препроцессора, на самом деле она предназначена для установок режимов компилятора и компоновщика (linker). В частности, для выравнивания по границе байта необходимо задать аргумент `pack(1)`.

Примечание

Система Borland C++ 3.1 этот аргумент "не понимает". Можно задавать выравнивание по границе байта так: `#pragma option -a-`. В более поздних системах это тоже работает.

Директива работает с момента объявления, поэтому поместив конструкцию

```
#pragma pack(1)
```

сразу после директивы `#include`, получим на выходе числа:

```
AAA 9
BBB 9
CCC 5
DDD 9
```

Однако, если расход памяти не критичен, можно об этом не беспокоиться.

ГЛАВА 4

Тяжелое наследие C

Одним из самых трудных понятий языка C++ для начинающего программиста являются указатели. Однако не разобравшись с ними, вы никогда не сможете программировать серьезные задачи. Например, вся компонентная технология основана на указателях. Указатели являются настолько фундаментальной конструкцией языка, что без них не было бы ни C, ни C++. Как заметил Д. Элджер [48], с помощью указателей реализуется одна из "великих идей C++" — косвенные обращения. *Косвенным обращением* называется ситуация, когда для доступа к некоторому объекту используется промежуточный объект. Указатели и являются этими промежуточными объектами. Как увидим далее, косвенность обычно повышает гибкость программы и способствует инкапсуляции.

В то же время с указателями связано и множество ошибок, причем такие ошибки зачастую оказываются самыми неприятными и труднообнаруживаемыми. Только в умелых руках указатели "ведут себя послушно и правильно", разрешая использовать всю мощь косвенных обращений. Новичок должен обращаться с указателями как "сапер с неразорвавшейся миной": одно неосторожное "движение" — и программа "самоликвидируется". Недаром в языке Java от явных указателей просто отказались.

Параметры-массивы в форме указателя

В *гл. 3* мы уже выяснили, что массивы передаются в функцию по указателю (см. листинг 3.2). Однако было бы совсем хорошо, если бы для любого массива `A` мы могли бы писать, например, такие вызовы функции суммирования:

```
Summa(A, A + 10)
Summa(A + 3, A + 8)
```

Первый вызов означает сумму первых 10-ти элементов массива. Второй, очевидно, — сумма 5-ти элементов, начиная с `A[3]` по `A[7]`. Как видим, такая функция суммирования позволяет вычислять любые частичные суммы массива. Она не получает лишних параметров — только то, что нужно для работы: начало и конец последовательности элементов массива. Мы можем написать подобную функцию, если используем способ задания параметров-массивов в форме указателей. Передача параметра-массива в форме указателя выглядит так:

```
тип *имя
```

Звездочка показывает, что это не простой параметр, а параметр-указатель. Таким образом, в заголовке функции суммирования мы должны написать два таких параметра. Но при этом имя массива в формальных параметрах не может быть задано — оно фигурирует только при вызове. Для большей универсальности напишем шаблон функции суммирования, текст которого приведен в листинге 4.1.

Листинг 4.1. Шаблон функции суммирования

```
template <class T>
T Summa(T *begin, T *end)
{  T sum = 0;
   if (begin < end)    //  если параметры корректны
   for (;begin < end; ++begin) sum += *begin;
   return sum;
}
```

Функция достаточно проста, но пока непонятна. Мы разберемся с особенностями реализации этой функции по мере изучения указателей. Здесь же обратим внимание только на оператор суммирования:

```
sum += *begin;    //  звездочка обязательна
```

Совершенно очевидно, что параметр `*begin` должен иметь такой же тип, как и переменная `sum`. Это в самом деле так, что мы и наблюдаем в заголовке.

Как видим, использование параметров-указателей действительно позволяет написать более универсальную функцию. Давайте разберемся, что это такое — указатели.

Что такое указатели

Как известно, память компьютера представляет собой массив. В процессоре Intel — это массив байтов. В массиве каждый элемент имеет индекс. Байты памяти тоже пронумерованы, начиная с нуля и до (n − 1). Этот номер на-

зывается *адресом*. Когда программа попадает в память компьютера для выполнения, каждая переменная и каждая функция размещаются по некоторому адресу. Каков реально этот адрес — нам знать вовсе не обязательно, важно, что каждое имя в программе на самом деле является адресом памяти. Говорят, что по этому адресу в памяти хранятся значения соответствующей переменной.

Однако легко себе представить ситуацию, когда по некоторому адресу находится не значение, а адрес другой переменной. Таким образом, в нашей программе на C++ может быть переменная, которая во время работы программы содержит адреса других переменных. Эта переменная и называется *указателем*, а описанная ситуация называется *косвенным обращением*. Опять же, какие это адреса — нам знать вовсе не нужно. Мы только должны правильно указать, что нам необходим адрес именно этой переменной, а не другой. Начнем с простейших объявлений:

```
int *p;    //  переменная p может содержать адрес объекта типа int
```

Переменная `p` является указателем на переменные целого типа. Эта конструкция принципиально отличается от конструкции

```
int p;    //  переменная p может содержать значение типа int
```

В последнем случае переменная `p` может содержать значение целого типа, а в первом — адрес переменной целого типа.

Чтобы присвоить указателю адрес переменной, в языке реализована *операция получения адреса*, которая обозначается символом `&` (амперсанд):

```
int a;
a = 3;      //  присвоение значения
int *pa;
pa = &a;    //  присвоение адреса
```

После выполнения операции присваивания переменная-указатель `pa` указывает на переменную `a`. Эта ситуация изображена на рис. 4.1.

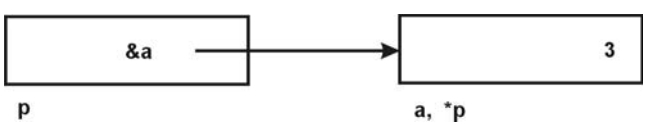

Рис. 4.1. Указатель `pa` указывает на переменную `a`

Этот же знак обозначает в C++ еще и битовую операцию И (and), однако для нас гораздо важнее является то, что при передаче параметра по ссылке тоже используется этот же символ. Более того, ссылки тоже можно объявлять, например:

```
int &ra = a;
```

Вообще-то говоря, амперсанд, используемый для обозначения ссылки, тоже означает получение адреса, однако это происходит на этапе трансляции. Именно поэтому со ссылками нельзя выполнять арифметических операций. Адрес "намертво приклеен" к ссылке, таким образом имя ссылки является синонимом имени переменной. А указатель является полноценной переменной, значение которой изменяется при выполнении программы.

Большинство сложностей при работе с указателями возникает из-за непонимания одной простой вещи: в объявлении

```
int *p;
```

переменная `p` является указателем и хранит адрес некоторой целой переменной, а конструкция `*p` — это целая переменная. Мы можем использовать данную конструкцию в любом месте программы, где допускается применение целой переменной. Именно таким образом мы использовали параметр-указатель `begin` в функции суммирования (см. листинг 4.1).

Вообще-то говоря, звездочка при имени переменной — это *операция разыменования*, которая используется для доступа к объекту, на который указывает указатель. Таким образом, операции `&` (получение адреса) и `*` (разыменования) являются парными и должны взаимно компенсироваться. Продемонстрируем эти операции на примере простой программы, текст которой приведен в листинге 4.2.

Листинг 4.2. Простейшие операции с указателями

```
#include <iostream.h>
int main()
{  int a = 5, b = 0;                  //  1
   int *pa; pa = &a;                  //  2
   cout << "a =" << a << endl;        //  3
   cout << "*pa =" << *pa << endl;    //  4
   cout << "pa =" << pa << endl;      //  4.1
   b = *pa - 3;                       //  5
   cout << "b =" << b << endl;        //  6
   if (a == *pa) *pa = b;             //  7
   cout << "a =" << a << endl;        //  8
   cout << "*pa =" << *pa << endl;    //  9
   b = 6; pa = &b;                    //  10
   cout << "b =" << b << endl;        //  11
   cout << "*pa =" << *pa << endl;    //  12
   cout << "a =" << a << endl;        //  13
   return 0;
}
```

Эта программа выведет на экран следующее:

```
a = 5              //  вывод оператора 3
*pa = 5            //  вывод оператора 4
pa = 0x0066FDE8    //  вывод оператора 4.1
b = 2              //  вывод оператора 6
a = 2              //  вывод оператора 8
*pa = 2            //  вывод оператора 9
b = 6              //  вывод оператора 11
*pa = 6            //  вывод оператора 12
a = 2              //  вывод оператора 13
```

Операторы `1` и `2` — это просто объявления переменных с инициализацией. Указатель `pa` получает адрес переменной `a`. Затем операторы `3` и `4` показывают нам, что `a` и `*pa` имеют одно и то же значение. Для демонстрации того, что `pa` и `*pa` — не одно и то же, в программу добавлена строка `4.1`. Адрес выводится на экран в шестнадцатеричной форме. Потом в строке `5` мы вычисляем и выводим значение переменной `b`. Выражение

```
*pa - 3
```

не изменяет указатель `pa`: от значения переменной `a` (на нее указывает `pa`) отнимается `3`, и результат записывается в переменную `b`. Следующий оператор `if` и операторы вывода в строках `8` и `9` демонстрируют нам неразрывную связь `a` и `*pa`: изменение `*pa` автоматически означает изменение `a`. Оператор

```
pa = &b;
```

в строке `10` разрывает эту связь указателя `pa` с переменной `a` и устанавливает новую связь — с переменной `b`. Операторы вывода `11-13` демонстрируют нам это.

Виды указателей

Теперь, когда мы немного попробовали указатели "на вкус", углубимся в некоторые детали. Прежде всего отметим, что в языке C++ существует три вида указателей (хотя с точки зрения реализации все они представляют одно и то же — адрес памяти):

- типизированный указатель на переменную `тип *имя`. Тип может быть как встроенным, так и определенным программистом;
- бестиповый указатель `void*`;
- указатель на функцию.

Наиболее часто используются типизированные указатели. С ними разрешается выполнять арифметические действия. Бестиповый указатель от типизи-

рованного отличается тем, что с таким указателем нельзя производить никаких арифметических операций. Однако с помощью преобразования типа бестиповый указатель всегда можно преобразовать в типизированный. Обычно бестиповые указатели применяются как параметры функций для повышения степени универсальности.

А указатель на функцию — это вообще особый вид указателя, который применяется исключительно с одной целью — косвенного вызова функции. С таким указателем тоже нельзя выполнять никаких арифметических действий.

Объявление типизированных указателей

Общий вид объявления типизированного указателя следующий:

```
тип *имя
```

Как видим, это ничем не отличается от объявления параметра в указательной форме, которое мы использовали при объявлении параметров-массивов (см. листинг 4.1). Как обычно, в одном операторе можно объявлять несколько указателей, можно смешивать объявления указателей и не указателей. Однако, как обычно это бывает в C++, программиста здесь подстерегают маленькие ловушки. Пусть, например, в программе есть следующее объявление:

```
int* pa, pb;
```

В этой записи переменная `pa` является указателем, а переменная `pb` — нет. Отсюда следует, что свойство "быть указателем" — это свойство переменной, а не типа.

Указатели можно инициализировать адресом другой переменной того же типа:

```
int a = 3;        //  инициализация целой переменной a
int *pa = &a;     //  инициализация указателя pa адресом a
int *h(&a);       //  инициализация указателя h адресом a
int *r = h;       //  инициализация другим указателем
```

Указатель `pa` получил значение — адрес переменной `a`. Точно тот же адрес получил и указатель `h`, и указатель `r`. Теперь в программе `*h`, `*pa` и `*r` означают одно и то же: значение переменной `a`, на которую указывают все указатели.

Главная причина, по которой происходит большинство "аварий" программ, — отсутствие адреса в указателе. Если мы не присвоим указателю никакого адреса, то при выполнении программы там окажется "мусор". Поэтому попытки использовать такой указатель, как правило, приводят к непредсказуемому поведению программы. Чтобы быть уверенным в правиль-

ной работе программы, необходимо всегда присваивать значение указателям. По крайней мере, обнулите указатель, чтобы потом можно было проверить его на ноль:

```
double *p = 0;
```

Проверка делается обычным образом с помощью логического отрицания:

```
if (!p). . .
```

Это часто помогает уберечь программу от "самоликвидации". До принятия стандарта вместо нуля надо было писать макрос `NULL`, после принятия стандарта можно писать и то и другое.

Как и другие переменные, можно объявлять константы-указатели, однако и тут нас поджидают сюрпризы:

```
int i;
const int ci = 5;
const int *pi = &ci;
```

При такой записи указатель `pi` не является константой — он указывает на константу целого типа. Правильное объявление константы-указателя пишется так:

```
int *const pi = &ci;    //  ci — константа
```

Однако такое объявление не транслируется! В этой записи указатель у нас константный, но указывать должен на переменную. Поэтому транслятор требует вместо адреса константы присвоить адрес переменной. Таким образом, когда мы объявляем указатели, у нас появляются следующие варианты объявлений:

- не константный указатель на не константу `тип *` — можно изменять и указатель, и переменную;
- не константный указатель на константу `const тип *` или `тип const *` — указатель изменять можно, а переменную нельзя;
- константный указатель на не константу `тип * const` — переменную изменять можно, а указатель нельзя;
- константный указатель на константу `const тип const *` или `const тип * const` — нельзя изменять ни переменную, ни указатель.

Ничто не запрещает нам объявлять указатели на указатели, например:

```
int **p;
```

Значением такой переменной должен быть адрес указателя — получается двойное косвенное обращение. Переменные "указатель-на указатель-на..." обычно используются при работе с многомерными массивами.

Бестиповые указатели и преобразование типов

Бестиповый указатель объявляется аналогично типизированному:

```
void *p;
```

Так как в C++ нет переменных типа `void`, присвоение адреса некоторой переменной указателю такого вида так просто не получится — необходимо сделать явное преобразование типа. Мы уже знаем *(см. гл. 1)*, что в C++ имеется несколько вариантов задания явного преобразования типа: по наследству от C остались:

```
(тип) выражение
тип (выражение)
```

Теперь понятно, что первая несколько необычная форма была включена в язык именно для преобразования указателей:

```
int a = 6;
int *pa = &a;
void *vpa = (void *)&a;     //  преобразование int* в void*
void *vp = (void *)pa;      //  преобразование int* в void*
```

Аналогично делается и обратное преобразование:

```
int a = 6;
int *pa;
void *vpa = (void *)&a;     //  преобразование int* в void*
pa = (int *)vpa;            //  преобразование void* в int*
```

Преобразования типизированных указателей из одного типа в другой выполняются точно таким же образом:

```
int a = 6;
double *dp = (double *)&a;
int *ip = (int *)dp;
```

Однако, как мы помним *(см. гл. 1)*, в стандарт C++ добавлены новые формы преобразования типов:

```
static_cast<тип>(выражение)
reinterpret_cast<тип>(выражение)
```

Оказывается, что с указателями `void *` работают обе формы:

```
int a = 6;
int *pa = &a;
void *pointer = static_cast<void *>(&a);
pointer = reinterpret_cast<void *>(pa);
```

А вот преобразования типизированных указателей можно делать только вторым способом — `static_cast` в этом случае не работает:

```
int a = 6;
int *pa = &a;
double *ddp = static_cast<double *>(pa);        //  ошибка трансляции
int *ip = static_cast<int *>(dp);               //  ошибка трансляции
double *dp = reinterpret_cast<double *>(&a);    //  работает
```

Visual C++ 6 выдает сообщение об ошибке `C2440` о невозможности преобразовать один указательный тип в другой.

И наконец, разберемся, сколько памяти занимают указатели разного типа. Для этого, как обычно, выполним простую программу, текст которой приведен в листинге 4.3.

Листинг 4.3. Память для указателей

```
int main()
{  cout << sizeof(void *) << endl;
   cout << sizeof(char *) << endl;
   cout << sizeof(unsigned char *) << endl;
   cout << sizeof(sort *) << endl;
   cout << sizeof(unsigned short *) << endl;
   cout << sizeof(int *) << endl;
   cout << sizeof(unsigned int *) << endl;
   cout << sizeof(long *) << endl;
   cout << sizeof(unsigned long *) << endl;
   cout << sizeof(float *) << endl;
   cout << sizeof(double *) << endl;
   cout << sizeof(long double *) << endl;
   return 0;
}
```

Если мы выполним эту программу в Visual C++ 6 или Borland C++ Builder 6, то получим во всех случаях одну и ту же величину — `4` байта. Это совпадает с нашими выводами при программировании функции суммирования массива (см. листинг 3.2). В более старой системе Borland C++ 3.1 все зависит от *модели памяти*. Однако мы не будем разбираться в этих тонкостях, т. к. это относится к работе в MS DOS.

Массивы и указатели

Для всех встроенных типов в C и C++ существуют константы соответствующих видов. Указатели — тоже встроенный тип, поэтому встает вопрос о том, как представить в языке константу-указатель. При разработке языка C

авторы приняли гениальное решение: константой-указателем является имя массива. Пусть у нас в программе объявлен массив и указатель:

```
int a[10];
int *p;
```

Тогда присвоить адрес массива переменной-указателю можно одним из следующих совершенно равноправных способов:

```
p = &a[0];
p = &a;
p = a;
```

С типизированными указателями разрешены арифметические действия, поэтому после одного из таких присваиваний выражение `p + i` означает адрес `i`-го элемента массива. Обратите внимание, что `i` — это номер элемента массива, а не количество байт. Транслятор правильно вычисляет адрес нужного байта, умножая `i` на `sizeof(тип)`. Значение `i`-го элемента можно получить, применив уже известную нам операцию разыменования:

```
*(p + i)
```

Скобки поставлены, поскольку приоритет операции разыменования `*` больше приоритета операции сложения `+`. Эта запись эквивалентна записи `p[i]` и имеет смысл `a[i]`. Выражение `*p + i` имеет совершенно другой смысл: `a[0] + i`. Так же, как и `a[i]`, выражение `*(p + i)` можно использовать и слева, и справа от знака присваивания.

Так как имя массива тоже является указателем, его тоже можно использовать в таких выражениях, например в `*(a + i)`, что означает `a[i]`. Только надо помнить, что имя массива — это все-таки константа, а константе нельзя присвоить другое значение (в данном случае значением является адрес другой переменной). Вообще C++ для массивов позволяет совершенно экзотические формы записи одного и того же выражения. Например, выражение `*(a + i)` эквивалентно выражению `*(i + a)` — от перемены мест слагаемых сумма не меняется, — а это можно записать как `i[a]`. Пусть переменная `i` равна `2`. Тогда при записи

```
cout << 3[a] << endl;
cout << (i + 1)[a] << endl;
cout << i[a + 1] << endl;
cout << * (i + a + 1) << endl;
```

во всех случаях на экран будет выведено значение `a[3]`.

Если указатель `p` указывает на элемент массива, то операция инкремента `p++` или `++p` "передвинет" указатель на следующий элемент массива: к указателю прибавляется `sizeof(тип)` байтов.

Пусть, например, указатель p указывает на 5-й элемент массива. Тогда выражения

```
*p++      //  1
(*p)++    //  2
*(p++)    //  3
```

имеют, как обычно, совершенно разный смысл:

- выражение в строке 1 позволяет выдать значение 5-го элемента массива и "передвинуть" указатель;
- выражение в строке 2 позволяет увеличить значение 5-го элемента массива;
- выражение в строке 3 позволяет "передвинуть" указатель и выдать значение 6-го элемента массива.

Как вы уже поняли, к указателю всегда можно добавить выражение целого типа или отнять. Однако операции умножения и деления с указателями не допускаются. Складывать два указателя тоже нельзя. А вот разность указателей — это важная операция. Пусть у нас в программе есть следующие объявления:

```
int a[10];
int *p1 = &a[1];
int *p2 = &a[8];
```

Тогда оператор вывода

```
cout << p2 - p1 << endl;
```

выведет на экран число 7 — количество элементов массива между a[1] и a[8].

Указатели как параметры

Указатели, как и любые другие переменные, можно передавать в качестве параметра — это мы уже видели на примере массивов. Такой способ остался по наследству от C и применяется в C++ в том случае, если функция должна в процессе своей работы изменить значения нескольких переменных вне функции. Рассмотрим уже традиционный пример — функцию Swap, обменивающую значения двух переменных. С параметрами-указателями это определение выглядит так:

```
void Swap(int *a, int *b)           //  указатели в заголовке
{ int t = *a; *a = *b; *b = t; }    //  обмен значений по указателям
```

В этой функции происходит обмен значений, на которые указывают заданные параметры-указатели a и b. Однако сами указатели передаются по значению.

При обращении к функции в качестве аргументов указываются адреса переменных (или адресные выражения — обычные выражения использовать нельзя), что выглядит с точки зрения синтаксиса не очень красиво:

```
Swap(&x, &y);
```

Используя этот способ, мы достигаем такого же эффекта, как и при передаче по ссылке, однако способ вызова функции несколько неуклюж и часто приводит к ошибкам, поскольку программист забывает указать символ `&` перед аргументом (довольно распространенная ошибка начинающих программистов).

Иногда в функциях требуется оперировать не значениями, а самими указателями. Для этого мы можем использовать два способа:

- передачу параметра-указателя по ссылке;
- передачу параметра-указателя по указателю.

Вариант функции `Swap` с передачей указателя по ссылке приведен в листинге 4.4.

Листинг 4.4. Передача указателя по ссылке

```
void Swap(int *&v1, int *&v2)
{ int *t = v2; v2 = v1; v1 = t; }
```

Объявление `int *&v1` должно читаться справа налево: `v1` является ссылкой на указатель на объект типа `int`. Внутри функции для обмена значений объявляется указатель. Вызов функции `Swap` делается так:

```
int i = 10;
int j = 20;
int *pi = &i;
int *pj = &j;    //  присвоение адресов
Swap(pi, pj);
```

Вариант функции `Swap` с передачей параметров-указателей по указателю приведен в листинге 4.5.

Листинг 4.5. Передача указателя по указателю

```
void Swap(int **v1, int **v2)
{ int *tmp = *v2; *v2 = *v1; *v1 = tmp; }
```

От звездочек начинает рябить в глазах! Да и при вызове тоже требуется прописывать "лишние" амперсанды:

```
Swap(&pi, &pj);
```

Фокус с несколькими выражениями вместо одного параметра проходит и для указателей — только последнее выражение в списке должно быть адресного типа.

Бестиповые указатели как параметры

Обычно слово `void` в списке параметров указывается тогда, когда параметров нет. Единственным исключением из этого правила является передача бестипового указателя `void *` в качестве параметра:

```
void f(void *p)
```

Аналогично, функция может возвращать бестиповый указатель:

```
void *f(void *p)
```

Бестиповый указатель применяется в тех случаях, когда тип указываемого значения может меняться от вызова к вызову при выполнении программы. Такие функции еще называются *полиморфными*.

Рассмотрим использование бестиповых указателей на простом примере перевода чисел в строку символов. Функция должна уметь переводить числа любого типа. Возникает проблема с типом переводимого числа: параметр-то единственный, а типы при вызове должны быть разные. Этого можно добиться единственным способом — передавать в качестве параметра указатель `void *`. Тогда нам нужен еще один параметр, указывающий тип числа, т. к. по типу указателя мы никак не можем этот тип определить. В теле функции придется писать оператор `switch`, в котором и будет выполняться обработка конкретного типа параметра. Текст функции перевода чисел приведен в листинге 4.6.

Листинг 4.6. Универсальная функция перевода чисел

```
enum Type { Int, Long, UnsignedLong, Float, Double };
int NumberToASCII(Type t, void *n, char buffer[])
{  switch(t)
   {  case Int: { int k = *(int *) n;       //  "обман" компилятора
      buffer = _itoa(k, buffer,10); break; }
      case Long: { long k = *(long *) n;    //  "обман" компилятора
      buffer = _ltoa(k, buffer,10); break; }
      case UnsignedLong:
      {  unsigned long k = *(unsigned long *) n;
         buffer = _ultoa(k, buffer,10); break; }
         case Float: { double k = *(float *) n;    //  "обман"
                                                   //  компилятора
         buffer = _gcvt(k, 4, buffer); break;
      }
```

```
      case Double: { double k = *(double *) n;      //  "обман"
                                                    //   компилятора
      buffer = _gcvt(k, 16, buffer); break; }
      default: cout << "Wrong types" << endl;
      return -1;
   }
   return 0;
}
```

Для реализации "многотиповости" нами объявлен перечислимый тип `Type`, который является первым параметром и служит переключателем в операторе `switch`. Вторым параметром является указатель на нашу числовую переменную, а третий параметр — символьный массив-строка, в котором сохраняется результат преобразования. Мы предполагаем, что вызывающая программа корректно этот массив объявила. В каждом операторе `case` сначала выполняется доступ к переменной соответствующего типа — здесь приходится "обманывать" компилятор с помощью преобразования типа. Далее вызывается соответствующая функция, осуществляются преобразования, прототипы которых прописаны в библиотеке stdlib.h. Использование данной функции демонстрирует пример, текст которого приведен в листинге 4.7.

Листинг 4.7. Использование функции `NumberToASCII`

```
int main(void)
{  int m = -123;
   long mm = -321;
   unsigned long mu = 1234567890;
   float d = -123.4567e-9;
   double dd = -123.45678901234;
   char ch[50];
   NumberToASCII(Int, &m, ch);
   cout << ch << endl;
   NumberToASCII(Long, &mm, ch);
   cout << ch << endl;
   NumberToASCII(UnsignedLong, &mu, ch);
   cout << ch << endl;
   NumberToASCII(Float, &d, ch);
   cout << ch << endl;
   NumberToASCII(Double, &dd, ch);
   cout << ch << endl;
   return 0;
}
```

С помощью механизма перегрузки это делается значительно элегантнее. Прототипы функций могут выглядеть так (параметр-тип уже не нужен):

```
int NumberToASCII(int n, char buffer[]);
int NumberToASCII(long n, char buffer[]);
int NumberToASCII(unsigned long n, char buffer[]);
int NumberToASCII(float n, char buffer[]);
int NumberToASCII(double n, char buffer[]);
```

Для сравнения приведем реализацию первой функции:

```
int NumberToASCII(int n, char buffer[])
{ buffer = _itoa(k, buffer, 10); }
```

Все значительно упростилось, поскольку исключаются лишние проверки, а вся обработка конкретного типа локализуется в отдельной функции. Кроме того, при отсутствии перегрузки определение типа преобразуемого аргумента осуществляется на этапе выполнения программы, а механизм перегрузки работает на этапе трансляции и не требует накладных расходов при выполнении программы.

Указатели на символы

В любой функции, в которой в качестве параметра передается массив символов, этот параметр можно передавать как указатель. Например, наша функция вычисления длины строки будет такой:

```
unsigned int Length(const char *s)
{ int L = 0; while (s[L++]); return (L - 1); }
```

Использование указательной формы позволяет задать значение по умолчанию для одномерных символьных массивов-строк. Как и для обычных переменных, в этом случае допускается присваивать в качестве значения по умолчанию строковую константу, например:

```
void Print(char *s = "Значение по умолчанию")
{ cout << s << endl; }
```

В системе Visual C++ 6 помимо указательной можно задавать "скобочную" форму параметра:

```
void Print(char s[] = "Значение по умолчанию")
```

Указатель в качестве возвращаемого значения позволяет нам возвращать из функции константу-строку. Б. Страуструп [37] приводит следующую функцию, которая корректно работает во всех системах:

```
const char* error_message(int i)
{ return "String of return"; }
```

В заголовке прописано `const`, однако оно не обязательно. Параметр `i` здесь не используется, поскольку функция просто иллюстрирует возврат константы-строки.

Функция, возвращающая указатель, конечно может быть использована для присвоения начального значения параметру-указателю.

Русские буквы

В *гл. 2* мы уже упоминали, что русские символы, вводимые с клавиатуры, правильно выводятся на экран во всех системах. А вот константы с русскими буквами верно отображаются только в системе Borland C++ 3.1. Проблема — в кодировке символов. Набирая программу в редакторе, мы работали в Windows, поэтому русская строка записалась в программе в кодировке, принятой в Windows. Работает же наша программа в консольном окне, где Windows использует кодировку ASCII, динамически выполняя перекодировку. Значит, наша задача — перекодировать нашу строку из Windows-кодировки в ASCII. Простейший способ добиться этого — использовать функцию WinAPI `CharToOem`. Прототип этой функции приведен в листинге 4.8.

Листинг 4.8. Определение функции перекодировки

```
BOOL CharToOem(
  LPCTSTR lpszSrc    //  указатель на преобразуемую строку
  LPSTR lpszDst      //  указатель на буфер для преобразованной строки
              );
```

Параметры:

- `lpszSrc` — указывает на завершающуюся нулем преобразуемую строку;
- `lpszDst` — указывает на массив, куда помещается преобразованная строка.

Функция всегда возвращает ненулевое значение.

Несмотря на то, что типы параметров указаны в нотации Windows, мы можем вместо этих малопонятных параметров подставлять символьные массивы. Причем, первый параметр — это наша строка, а второй — это символьный массив, куда заносится перекодированная строка. Первый и второй параметры могут быть одним и тем же символьным массивом. Чтобы использовать эту функцию, мы должны подключить заголовок `windows.h`. Простейший вариант функции, перекодирующей символьные массивы, приведен в листинге 4.9.

Листинг 4.9. Перекодировка символьных массивов

```
char * Rus(char in[], char out[])
{ if (CharToOem(in, out)) return out; else return 0; }
```

Естественно, заголовок мы можем написать с параметрами в указательной форме:

```
char * Rus(char *in, char *out)
```

Возврат указателя позволяет нам использовать вызов этой функции непосредственно в операторе вывода:

```
char s[100] = "Привет от строки!";
cout << Rus(s, s) << endl;
```

Эта же функция позволит нам выводить на экран строки-константы:

```
char s[100];
cout << Rus("Снова привет! Большой и горячий! Hello!", s) << endl;
```

Библиотека string.h

Серьезная обработка строк требует реализации разнообразных операций: получение заданной подстроки из строки, сравнение строк, удаление заданной подстроки из строки или замена на другую последовательность символов. Набор подобных операций реализован в виде стандартной библиотеки string.h. Библиотека входит в стандарт C и по наследству досталась C++. По сложившейся традиции в функции обработки символьных массивов-строк передаются указатели (`char *`). Например, прототип функции, вычисляющей длину строки, выглядит следующим образом:

```
size_t strlen(const char *s);
```

где параметр `size_t` определен с помощью оператора `typedef` как `unsigned int`.

Некоторые наиболее часто используемые функции приведены в табл. 4.1. Надо помнить, что эти функции ориентированы исключительно на английский алфавит, представленный в ASCII-коде. С русским языком при использовании некоторых функций возникают проблемы. Русифицируем функции `strlwr` и `strupr`, меняющие регистр строки-аргумента, но не будем связываться с явными кодами символов. Прототипы наших функций отличаются от библиотечных только именами, сохраняя и список параметров, и тип возвращаемого значения. Текст примера русификации функции `strlwr` приведен в листинге 4.10.

Листинг 4.10. Русификация функции `strlwr`

```
char *strRlwr(char *s)
{  const int nA = 33;    //  место для нуля
   static const char RusB[nA] = "АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
   static const char RusM[nA] = "абвгдежзийклмнопрстуфхцчшщъыьэюя";
```

```
   int L = strlen(s);
   for (int i = 0; i < L; ++i)
   for (int j = 0; j < nA; ++j)
   if (s[i]==RusB[j]) { s[i] = RusM[j]; break; }
   strlwr(s);
   return s;
}
```

Функция `strRupr` отличается от `strRlwr` только одним оператором:

```
if (s[i] == RusM[j]) { s[i] = RusB[j]; break; }
```

В остальном две функции идентичные. И тут нам пригодились статические переменные: массивы `RusB` и `RusM` будут созданы один раз при первом обращении к функциям.

Таблица 4.1. *Функции обработки символьных массивов*

Прототип	Описание
`char *strcat(char *dst, const char *src)`	Прицепляет строку `src` к строке `dst`. Длина результата равна `strlen(src + dst)`
`char*strncat(char*dst, const char*src, size_t n)`	Прицепляет `n` первых символов строки `src` к строке `dst`. Длина результата равна `strlen(dst) + n`
`char *strchr(const char *s, int c)`	Поиск символа `c` в строке `s` слева направо
`const char *strrchr(const char *s, int c)`	Поиск символа `c` в строке `s` справа налево
`const char*strstr(const char*s1, const char*s2)`	Поиск строки `s2` в `s1` слева направо
`int strcmp(const char *s1, const char *s2)`	Сравнение строк как массивов байтов (беззнаковые целые — коды символов). Если `s1 < s2`, то результат `< 0`. Если `s1 = s2`, то результат `== 0`. Если `s1 > s2`, то результат `> 0`
`int stricmp(const char *s1, const char *s2)`	Сравнение строк как массивов байтов без учета регистра
`int strncmp(const char*s1, const char*s2, size_t n)`	Сравнение `n` первых символов строк как массивов байтов
`char *strcpy(char *dst, const char *src)`	Копирует строку `src` в строку `dst`

Таблица 4.1 *(окончание)*

Прототип	Описание
`char *strncpy(char *dst, const char *src, size_t n)`	Копирует `n` символов из `src` в строку `dst`
`char *strlwr(char *s)`	Преобразует прописные буквы в строчные в строке `s`
`char *strupr(char *s)`	Преобразует строчные буквы в прописные в строке `s`

А теперь вспомним определение структуры `Person` *(см. гл. 3)*. Мы можем присвоить значение полю — символьному массиву с помощью функции `strcpy`:

```
strcpy(mans[5].fio, "Фамилия");
```

Указатели на символы — переменные и константы

В любой системе фирмы Borland наши функции прекрасно работают. Однако при проверке функций в системе Visual C++ 6 выясняется, что не всегда они работают так, как нам хотелось бы. Например, оператор

```
cout << Rus(strRlwr("ПРИВЕТ!"), ss) << endl;
```

не работает, а вот такой вариант

```
char ss[100] = "ПРИВЕТ!";
cout << Rus(strRlwr(ss), ss) << endl;
```

выполняется правильно. Получается, что передача константы-строки и массива символов выполняется по-разному, несмотря на то, что тип формального параметра для них одинаков (`char *`). Для того чтобы это выяснить, рассмотрим два объявления с инициализацией:

```
char *s1 = "Указатель на символы";
char s2[] = "Массив символов";
```

Несмотря на то, что при передаче параметров эти две формы эквивалентны, в данном случае разница довольно значительна:

- `s1` — это указатель, который занимает в памяти только 4 байта; `s2` — это массив символов, который занимает в памяти 16 байт (15 символов и нулевой байт);

- `s1` указывает на неизменяемую строку — ни один символ в этой строке мы изменить не сможем; `s2` — это обычный массив символов, любой из которых мы можем изменить, например `s2[1] = 'П'`;
- разрешено присваивание `s1 = s2`; и не разрешено `s2 = s1`.

Таким образом выясняется, что если мы передаем строку-константу, то изменить ее не сможем. Именно поэтому наши функции "не работали". А компиляторы фирмы Borland и в этом случае нарушают стандарт.

Как для числовых, так и для символьных массивов C++ разрешает необычные формы обращения, например

```
cout << ("abcdef" + 2)[0] << endl;
cout << 2["abcdef"] << endl;
```

и даже просто

```
cout << "abcdef" + 2 << endl;
```

выведут на экран символ "c". Понятно, что на самом деле в операциях участвует не сама константа-строка, а неявный указатель на нее.

В *гл. 3* мы использовали двумерный массив символов для записи названий дней недели. Обычно для таких случаев используется массив указателей на строки:

```
char *Days[] = {"Воскресенье", "Понедельник", "Вторник", "Среда",
                "Четверг", "Пятница", "Суббота"
               };
```

Как видите, нам нет необходимости задавать длину одной строки (количество элементов символьного массива). В результате компилятор эту длину вычисляет и резервирует для строк именно столько байтов, сколько необходимо. Пусть в программе объявлен двумерный массив и массив указателей на символы:

```
char s[3][10] = {"имя", "строка", "типы"};
char *s[10] = {"имя", "строка", "типы"};
```

Разницу между двумерным символьным массивом и массивом указателей на строки демонстрируют рис. 4.2 и 4.3.

Для большого количества строк, некоторые из которых длинные, но большинство коротких, получается значительная экономия памяти, даже несмотря на наличие массива указателей.

Напишем функцию, текст которой приведен в листинге 4.11, которая выведет этот массив строк на экран, используя в качестве парамера массив указателей на строки. Это стандартный прием при программировании функций обработки строк. Количество строк необходимо передавать функции в качестве параметра, как и в случае обычных одномерных массивов.

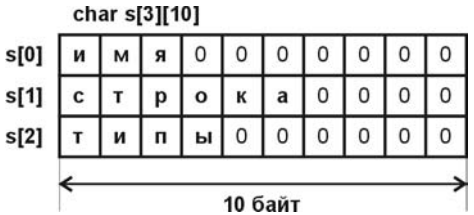

Рис. 4.2. Двумерный массив символов

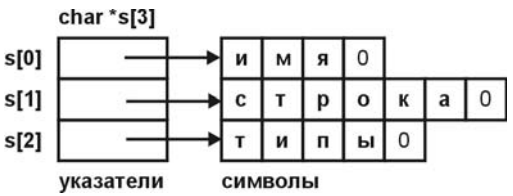

Рис. 4.3. Массив указателей на строки

Листинг 4.11. Функция вывода массива строк на экран

```
void PrintArrayString(char *m[], int n)
{ for (i = 0; i < n; ++i) cout << Days[i] << endl; }
```

Однако для параметра-массива указателей мы можем упростить заголовок, если будем придерживаться того же соглашения, что и для строк: последний элемент массива указателей должен иметь нулевое значение. Тогда отпадает необходимость передавать длину массива. Ту же программу можно записать так, как приведено в листинге 4.12.

Листинг 4.12. Передача массива указателей без количества элементов

```
void PrintArrayString(char *m[])       //  нет количества элементов
{  for (i = 0; Days[i]; ++i)           //  указатель не ноль?
   cout << Days[i] << endl;
}
int main(void)
{  char *s[] =                         //  массив указателей на строки
{  "First", "Second", "Three", 0 };    //  нулевой указатель
   PrintArrayString(s);                //  вызов функции
   return 0;
}
```

Этот же прием может использоваться не только для строк, но и для любых двумерных (или многомерных) массивов.

Указатели и динамическая память

Все вышесказанное, тем не менее, не объясняет необходимость наличия указателей в C++ — в конце концов, многие языки программирования обходятся без них. Один довод в пользу указателей нам известен: чтобы уменьшить накладные расходы при вызове функции, параметром которой является массив, в функцию передается только указатель, занимающий 4 байта. Однако есть значительно более весомая причина: указатели используются при работе с *динамическими переменными*.

Динамические переменные создаются и уничтожаются явным образом во время выполнения программы. Доступ к таким переменным осуществляется только с помощью указателей. Таким образом, "время жизни" динамической переменной не зависит от локальности и глобальности, а видимость определяется *областью видимости* указателя, который на нее указывает. Для создания динамической переменной используется операция `new`, для уничтожения — операция `delete`. Объявив в программе указатель

```
int *p;
```

мы можем создать динамическую переменную

```
p = new int;
*p = 10;
```

и удалить ее

```
delete p;
p = 0;
```

Обнулять указатель, конечно, необязательно, но если вы хотите уберечь себя от лишних проблем, лучше сделайте это сразу после удаления.

Динамические переменные можно инициализировать:

```
int *p = new int(1024);
```

Тип динамической переменной может быть любым, в т. ч. и определенным пользователем. Например, мы можем объявить указатель типа `Date` (см. листинг 3.14) и создать динамическую переменную:

```
Date *p = new Date;
```

Однако инициализировать такие переменные уже не получится — для этого в структуре должен быть реализован конструктор инициализации *(см. гл. 8)*. "Добраться" до полей динамической структуры можно двумя способами. Первый вариант — использовать селектор и операцию разыменования:

```
(*p).year = 2004;
```

Скобки ставить обязательно, т. к. приоритет операции "точка" выше приоритета операции разыменования. Второй способ доступа обеспечивается другой замечательной операцией разыменования ->:

```
p -> month = 12;
```

Замечательная она в том смысле, что ее можно перегрузить, и реализация так называемых "интеллектуальных" указателей основана именно на этом.

Операция `new` выделяет память, записывает адрес в указатель и, если необходимо, инициализирует объект. Заметим, что динамический объект не имеет явного имени и доступ к нему только косвенный — по указателю. Операция `delete` возвращает память системе (но не обнуляет указатель!). Если мы забудем использовать эту операцию, то возникает утечка памяти, которая может привести к аварийному завершению программы.

Чаще всего используются динамические массивы. Создание динамического массива выполняется операцией `new[]`, уничтожение, соответственно, операцией `delete[]`. А вот проинициализировать динамический массив нет возможности.

Примечание

По стандарту при создании массива в качестве размера допускаются только целочисленные значения, однако не все компиляторы придерживаются стандарта. Например, компиляторы от Borland "кушают" такую конструкцию, как `char *s = new char[sin(x)]`.

Кстати, нулевой размер динамического массива допускается.

Напишем функцию дублирования строки, текст которой приведен в листинге 4.13. Аналогичная функция есть в библиотеке string.h, однако мы рассмотрим свою, чтобы научиться работать с динамическим массивом.

Совершенно очевидно, что параметром функции должен быть указатель на строку символов. Функция должна создать динамический массив, переписать в него исходную строку и возвратить указатель на него.

Листинг 4.13. Функция дублирования строки с динамическим массивом

```
char *Duplicate(char *s)
{  char *ss = new char[strlen(s) + 1];     //  создаем динамический массив
   strcpy(ss, s);                          //  копируем строку
   return ss;                              //  возвращаем указатель
}
```

Подчеркнем, что память выделяется в функции, а возвращать ее системе должна вызывающая программа. Эта ситуация часто приводит к ошибке

утечки памяти, т. к. программист забывает освободить память, выделенную функцией.

Поскольку имя массива является указателем, при вызове функции разрешается в качестве фактического параметра задавать динамический массив с помощью оператора `new`. Но в функцию, естественно, передается только указатель, поэтому длину такого массива все равно придется задавать. Пример использования динамического массива как параметра приведен в листинге 4.14.

Листинг 4.14. Параметр — динамический массив

```
int ff(int m) { return m; }    //  функция для задания размера массива
int* f(int n, int *a)
{  //  что-то делаем с массивом
   return a;
}
int main(void)
{  int k = 3;
   int *p = f(k, new int[ff(k)]);    //  динамический массив
                                     //  как параметр
   delete []p;                       //  возврат памяти
   return 0;
}
```

Как видите, в качестве размера динамического массива можно задавать произвольное выражение, в данном случае — вызов функции. Необходимо подчеркнуть, что функция, получающая динамический массив в качестве параметра, обязана возвращать указатель, иначе возникает утечка памяти.

Многомерные динамические массивы

Если с одномерными массивами все более или менее понятно, то многомерные вызывают много вопросов. Повторим, что массивы в C++ бывают только одномерные, но элементами массива могут быть сами массивы. Чтобы разобраться в этом вопросе, давайте напишем классическую программу, которую часто приходится писать студентам — умножение матриц. Для упрощения будем считать, что матрицы квадратные. Пусть исходные матрицы имеют имена `A` и `B`, а результирующая — `R`. Напомним, что элемент результирующей матрицы `r[i][j]` вычисляется по формуле:

$$r_{ij} = \sum_k a_{ik} \times b_{kj}$$

Если нам известен размер матриц `n`, то элементы матрицы `R` вычисляются в трехкратном цикле:

```
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
{  float s = 0; for (int k = 0; k < n; k++) s += a[i][k] * b[k][j];
   r[i][j] = s;
}
```

Мы можем реализовать этот цикл для любого конкретного `n`: `3`, `5`, `7`, `17` и т. д. Но ведь нам желательно написать универсальную программу, поэтому без динамических массивов тут не обойтись, т. к. только для динамического массива мы можем задавать размер, вычисляемый во время выполнения программы. Однако и тут нас поджидают несколько сюрпризов — мы не можем объявить матрицу `n × n`:

```
float *r = new float[n][n];
```

Такую конструкцию ни один компилятор не пропустит, т. к. с помощью переменной может задаваться только один размер — самый левый. Эта проблема решается так: сначала создается динамический массив указателей на массивы, затем каждый указатель инициализируется динамическим массивом для чисел. Таким образом, наш "главный" указатель является указателем на указатели. В C++ выделение памяти выглядит так:

```
int n;    //  размер матрицы
cout << "Введите размер матрицы: ";
cin >> n;
float **m = new float *[n];    //  массив указателей на строки матрицы
for (int i = 0; i < n; ++i) m[i] = new float[n];    //  строки матрицы
```

Таким способом выделяется память для всех трех матриц. Потом исходные матрицы `A` и `B` заполняются значениями, и выполняется умножение по приведенной выше схеме. После выполнения работы полученную память необходимо вернуть системе, иначе возникает утечка памяти:

```
for (i = 0; i < n; i++) delete a[i];
delete []a;
```

Сначала возвращаются массивы для чисел, а затем — массив указателей. Текст полной версии программы приведен в листинге 4.15.

Листинг 4.15. Умножение матриц — динамических массивов

```
int main(void)    //  главная программа
{  int i, j, n;
   cout << "Введите размеры матриц A, B, R ";
   cin >> n;
// выделение памяти под матрицы
   float **a = new float*[n];     //  указатель на матрицу A
```

```
   for (j = 0; j < n; j++) a[j] = new float[n];
   float **b = new float*[n];     //  указатель на матрицу B
   for (j = 0; j < n; j++) b[j] = new float[n];
   float **r = new float*[n];     //  указатель на матрицу R
   for (j = 0; j < n; j++) r[j] = new float[n];
// ввод матриц A и B
   for (i = 0; i < n; i++)
   { for (j = 0; j < n; j++) cin >> a[i][j]; cout << endl; }
   cout << endl;
   for (i = 0; i < n; i++)
   { for (j = 0; j < n; j++) cin >> b[i][j]; cout << endl; }
   cout << endl;
// умножение матриц
   for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
   { float s = 0; for (int k = 0; k < n; k++)
     s += a[i][k] * b[k][j]; r[i][j] = s; }
// вывод на экран матрицы — результата
   cout << "Ответ" << endl;
   for (i = 0; i < n; i++)
   { for (j = 0; j < n; j++) cout << setw(6) << r[i][j]; cout << endl; }
   for (i = 0; i < n; i++) delete a[i]; delete []a;
   for (i = 0; i < n; i++) delete b[i]; delete []b;
   for (i = 0; i < n; i++) delete r[i]; delete []r;
   return 0;
}
```

Интересно, что эта программа правильно работает даже при задании размера матриц, равного `1` (единица). Совершенно верно вычисляется произведение двух чисел.

Многомерные массивы как параметры

Хорошо бы оформить нашу программу умножения матриц как ряд функций. Начнем с функции вывода матрицы на экран. Для упрощения пусть это будет квадратная матрица. Однако тут нас поджидают некоторые сложности, т. к. надо передавать в качестве параметра двумерный массив. Так же как и при инициализации, неопределенной может быть только одна размерность, поэтому следующий заголовок недопустим:

```
void PrintMatrix(int m[][], int n)
```

Поэтому в качестве параметра может быть задан либо массив указателей

```
void PrintMatrix(int *m[], int n)
```

либо двойной указатель на указатель

```
void PrintMatrix(int **m, int n)
```

И в том и в другом случае тело функции будет совершенно одинаковым:

```
for (int i = 0; i < n – 1; i++)
{ for(int j = i + 1; j < n; j++) cout << m[i][j]; cout << endl; }
```

Остальные функции оформляются совершенно аналогично.

На этом пока закончим экскурс в динамические массивы. Необходимо отметить, что с появлением в стандарте языка контейнеров библиотеки STL потребность в динамических массивах существенно снизилась.

Структуры и указатели

Структуры также можно передавать по указателю. В следующем примере, текст которого приведен в листинге 4.16, в функцию передается указатель на структуру. В самой функции показаны два разных способа доступа к элементам структуры.

Листинг 4.16. Передача структуры по указателю

```
struct Old { int a; float b; };
Old g(Old *p)                //  параметр – указатель
{  p -> a = p -> b + 6;      //  доступ к элементу операцией "->"
   (*p).b = p -> a + 7;      //  доступ к элементу селектором "."
   return *p;                //  возврат значения, а не указателя
}
void main(void)
{  Old y = { y.a = 1, y.b = 2 }, x;
   x = g(&y);                //  параметр – адрес структуры
}
```

Отметим, что вызов с передачей параметра-указателя изменяет значение исходной структуры. Вызов

```
x = g(&y);
```

приводит к тому, что значения переменных `x` и `y` становятся одинаковыми.

Структуры с указателями

Очень часто при решении различных задач мы не знаем, сколько элементов данных надо будет обрабатывать. Например, нам нужно написать программу, моделирующую обслуживание автомобилей на станции техобслужива-

ния. Мы не можем заранее предполагать, сколько автомобилей и в какие моменты времени появятся на станции. Понятно, что нам нужно реализовать очередь, элементом которой будет автомобиль.

Для представления автомобиля можно использовать структуру с тегом `Car`. При появлении автомобиля на станции мы создаем новую динамическую переменную, при убытии — уничтожаем ее. Осталось разобраться, как связывать неизвестное количество динамических переменных в очередь. Естественно, только с помощью указателей. Объявим новую структуру `Node` следующим образом:

```
struct Node { Car car_; Node *next; };
```

Такая запись допустима, несмотря на то, что описание структуры типа `Node` при объявлении типизированного указателя `Node *next` еще не завершено. Без звездочки тут, конечно, писать нельзя именно по причине "самовложенности".

Теперь мы можем объявить и переменные типа `Node`, и указатели типа `Node *`. Для реализации очереди нам потребуются указатели на начало и конец очереди. Каждый автомобиль, прибывший на станцию техобслуживания, реализуем динамической переменной типа `Node`, поэтому нам понадобится еще один указатель:

```
Node *Head, *Tail;     //  указатели на начало и конец очереди
Node *Cars;            //  прибывший автомобиль
```

Прибывший автомобиль мы прицепим в конец очереди, а уезжающий будем отцеплять от начала очереди. Именно для этого нам и нужен указатель в структуре: у последнего автомобиля указатель равен нулю, у предпоследнего указывает на последний, и т. д. Указатель первого указывает на второй, указатель второго — на третий, и т. д. А для доступа к первому и последнему автомобилю нам нужны два указателя `Head` и `Tail`. Эта ситуация представлена на рис. 4.4.

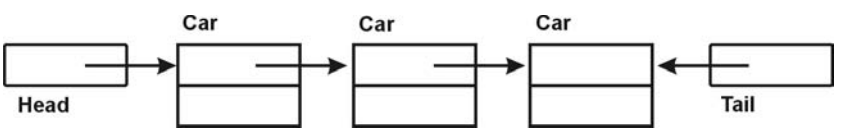

Рис. 4.4. Очередь автомобилей

Прибытие первого автомобиля реализуется так:

```
Cars = new Node;       //  автомобиль прибыл
Cars -> next = 0;      //  2
Head = Tail = Cars;    //  поставили в очередь
```

В строке `2` мы используем операцию `->` для доступа к указателю `next`.

Прибытие второго и последующих автомобилей реализуется так:

```
Cars = new Node;          //  автомобиль прибыл
Cars -> next = 0;         //  следующего нет пока
Tail -> next = Cars;      //  3, прицепили в хвост
Tail = Cars;              //  4, сделали последним
```

Строки `3` и `4` должны быть написаны именно в таком порядке, иначе автомобиль не "прицепится" в конец очереди. Отъезд автомобиля из очереди делается следующим образом:

```
Cars = Head;              //  сохранили доступ
Head = Head -> next;      //  следующий — первый
delete Cars;              //  освободили память
```

И опять манипулирование указателями требуется делать именно в таком порядке, иначе программа работать не будет.

Мы использовали связанные структуры для реализации очереди. Но точно таким же образом можно реализовать и стек. В самой организации связанных структур ничего не меняется. Стек от очереди отличается только дисциплиной доступа. Вставка и удаление элементов для стека выполняются только с одной стороны — с вершины стека. Удаление мы уже делали, поэтому покажем вставку:

```
Cars = new Node;          //  автомобиль прибыл
Cars -> next = Head;      //  прицепили список к новому
Head = Cars;              //  переставили указатель
```

И опять надо подчеркнуть, что присваивание указателей должно выполняться именно в указанном порядке, иначе возникают проблемы.

Структуры данных, подобные описанной очереди, называются *динамическими структурами данных*. В частности, наша очередь реализована в виде *линейного списка*. Один элемент списка называется *узлом* (node) и всегда представляется структурой, состоящей из двух частей: *информационной* и *указательной*. В зависимости от организации указательной части различают разные виды списков, которых очень много. Например, наша очередь реализована в виде *односвязного* линейного списка — т. к. указатель в структуре только один. *Кольцевой* список (рис. 4.5) характеризуется тем, что указатель последнего элемента показывает на первый элемент. Можно прописать в указательной части два указателя, один из которых указывает на следующий элемент, а второй — на предыдущий. Такой список называется *двусвязным* и представлен на рис. 4.6. Именно так реализован контейнер `list` в стандартной библиотеке STL.

Два указателя в узле могут играть и другие роли. Например, с помощью двух указателей реализуется *двоичное дерево*, представленное на рис. 4.7.

Информационная часть в этом случае содержит *ключ*. Один указатель показывает на узел с меньшим значением ключа, а второй — с большим значением. *Ассоциативные* контейнеры `map` (словарь) и `set` (множество) в стандартной библиотеке STL реализованы как один из видов *сбалансированного* двоичного дерева.

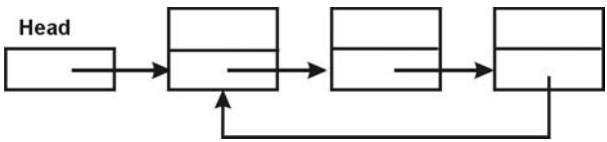

Рис. 4.5. Кольцевой список

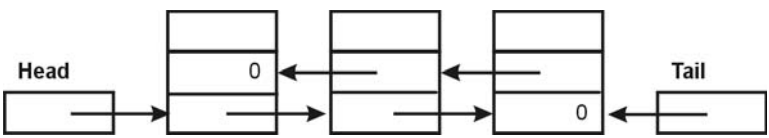

Рис. 4.6. Двусвязный список

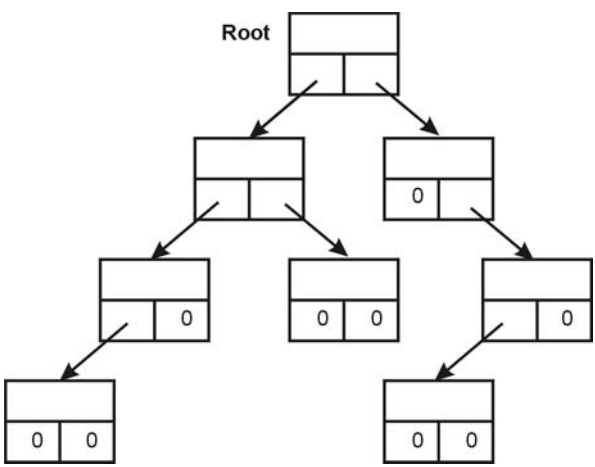

Рис. 4.7. Двоичное дерево

Если мы используем один указатель для связи со следующим узлом, а второй — как указатель на вложенный список, то получим новую структуру, которая называется *списком списков* (рис. 4.8). С помощью списка списков часто реализуются графы.

Структура узла с двумя связями одинакова во всех вариантах. Напишем обобщенный шаблон такой структуры:

```
template <class T>
struct Node
{  T info;    //  информационная часть
   first *Node;
   second *Node;
};
```

Данная структура позволяет нам реализовать и двусвязный список (который тоже может быть кольцевым), и двоичное дерево, и список списков. Однако имена указателей обычно меняются в зависимости от типа динамической структуры и ролей указателей. Для двоичного дерева указатели обычно имеют имена `Left` (левый) и `Right` (правый). Обычно первый узел дерева называется *корнем*. Узлы, на которые указывают левый и правый указатели, называются "сыновьями" (или *дочерними узлами*). Левый узел обычно называют "младшим сыном", а правый — "старшим сыном". Если "сыновей" нет, то узел называется *листом (концевой узел)*.

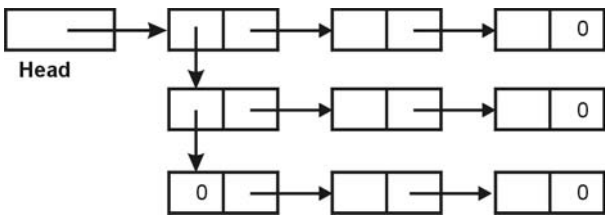

Рис. 4.8. Список списков

До изобретения шаблонов в информационной части тоже использовался указатель:

```
void *info;
```

Динамические структуры данных были придуманы для решения проблем с памятью. В литературе [5, 9, 14, 16, 34, 39, 40, 41, 45, 47] динамические структуры описаны достаточно хорошо, поэтому более подробно мы на этом останавливаться не будем, тем более, что стандартные контейнеры позволяют обойтись без них.

Проблемы с указателями

С указателями так часто возникают аварийные ситуации, что просто необходимо знать о наиболее типичных из них. Проблема в том, что область видимости и "время жизни" самих указателей и объектов, на которые они указывают, могут отличаться. Ошибки возникают в двух случаях:

- когда объект еще существует, а указатель на него отсутствует;
- когда указатель еще "живет", а объект уже не существует.

Первая ситуация называется "потерянная ссылка", вторая — "висячая ссылка". Следующий пример (его текст приведен в листинге 4.17) демонстрирует работу "висячей ссылки".

Листинг 4.17. Аварийная ситуация: "висячая ссылка"

```
int *f(void)
{  int Lokal = 1;     //  локальная переменная
   return &Lokal;     //  возврат адреса локальной переменной
}
void main(void)
{  int a = 2;
   int *p = &a;
   cout << "*p =" << *p << endl;     //  выводит *p = 2
   p = f();
   cout << "*p =" << *p << endl;     //  выводит *p = 1
   *p = 3;
   cout << "*p =" << *p << endl;     //  выводит *p = 3
   a = *p;
   cout << "*p =" << *p << endl;     //  *p = 6684152
   cout << "a =" << a << endl;       //  a = 6684152
}
```

Формально определение функции правильно — компилятор не будет сильно "протестовать". Однако фактически функция чрезвычайно опасна и при ее использовании могут возникнуть непредсказуемые результаты. В этой программе указатель `p` после вызова функции `f` "повис", однако обнаруживается это не сразу: на экран выводится `*p = 1`. Далее нормально срабатывает операция присваивания `*p = 3`, что демонстрирует соответствующий оператор вывода. Однако после него вдруг оказывается, что `p` указывает неизвестно на что, и значение `*p` совсем не равно `3`!

Итак, "висячие ссылки" возникают тогда, когда исчезают данные, на которые ссылается указатель. Это могут быть не только локальные, но и динамические переменные: если два разных указателя указывают на одно и то же, и один из них удаляется, то второй "повисает".

"Потерянные ссылки" фактически означают утечку динамической памяти. Ситуацию иллюстрирует следующий пример, текст которого приведен в листинге 4.18.

Листинг 4.18. Аварийная ситуация: "потерянная ссылка"

```
void X1(void)
{ int *p = new int[1000]; }     //  локальный указатель
void main (void)
{ X1(); }                       //  утечка памяти
```

В приведенной программе динамическая память запрашивается внутри функции `X1`, и доступ к этой памяти осуществляется по локальному указателю `p`. Но при выходе из функции локальный указатель теряется и, тем самым, теряется доступ к динамическому массиву. Таким образом, память не может быть возвращена системе. Большое количество запросов памяти, не возвращаемой обратно, приведет к аварийной остановке программы.

Никогда нельзя забывать, что параметры-указатели передаются по значению, поэтому попытка получить из функции указатель на динамическую память через параметр запросто может привести к утечке памяти:

```
void X1(int *p)              //  параметр-указатель
{ p = new int[1000]; }       //  запрос памяти
void main(void)
{ int *pp = 0; X1(pp); }     //  утечка памяти
```

Ситуация совершенно аналогична ситуации с локальным указателем, описанной в листинге 4.18. Указатель `pp` не изменяется после вызова функции `X1`. Таким образом, в программе возникает утечка памяти. Чтобы получать значение указателя на динамическую память через параметр-указатель, необходимо передавать либо указатель на указатель `int **p`, либо ссылку на указатель `int *&p`.

Подобные ошибки чрезвычайно трудно находить, поскольку условия их возникновения могут меняться от запуска к запуску.

Указатели на функции как параметры

Давайте обобщим функцию умножения на 2 (см. листинг 3.6). Хотелось бы написать такую функцию, которая могла бы обрабатывать любую последовательность элементов массива, причем операцию обработки тоже хотелось бы задавать при каждом вызове. Операция обработки — это тоже функция, выполняющая модификацию одного элемента массива заданным образом. Наша "главная" функция должна уметь применить эту операцию к каждому элементу заданной последовательности.

Язык C++ разрешает передавать функции в качестве параметра. Для этого в заголовке "главной" функции на месте одного из параметров (лучше первым, или последним) надо прописать прототип функции-параметра. Очевидно, что параметром функции-параметра должен быть элемент массива, а результат — того же типа, что и элементы массива. Таким образом, мы можем написать функцию обработки любой последовательности элементов массива, используя в качестве параметров пару указателей, как в шаблоне `Summa` (см. листинг 4.1). Такая функция, естественно, называется `ForEach`. Проверять параметры не будем, чтобы не отвлекаться от главного. Текст шаблона "главной" функции приведен в листинге 4.19.

Листинг 4.19. Шаблон `ForEach`

```
template <class T>
void ForEach(T *begin, T *end, T f(T elem))
{ for (; begin < end; ++begin) *begin = f(*begin); }
```

Как видите, оператор цикла полностью совпадает с тем, что мы написали в шаблоне `Summa`. Поменялось только тело цикла. Это обычная картина при программировании: если исходные принципы хороши, то программирование существенно упрощается. Теперь мы определим реальные функции и проверим работу нашего шаблона:

```
int f2(int a)          //  удвоение параметра
{ return (a* = 2); }
double f3(double a)    //  +2 к параметру
{ return (a += 2); }
int main(void)
{  int a[10] = { 1, 2, 3, 4, 5, 16, 7, 8, 9, 10 };
   ForEach(a, a + 10, f2);
   for (int i = 0; i < 10; ++i) cout << a[i] << ' ';
   cout << endl;
   double b[10] = { 10, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
   ForEach(b, b + 10, f3);
   for (int i = 0; i < 10; ++i) cout << b[i] << ' ';
   cout << endl;
}
```

На экран выводится совершенно правильный результат в обоих случаях:

```
2 4 6 8 10 32 14 16 18 20
12 14 15 16 17 18 19 20 21 22
```

Таким образом, мы написали универсальную функцию для обработки любой последовательности однотипных элементов. В стандартной библиотеке шаблонов STL сделано почти то же самое в стандартных алгоритмах.

С параметрами-функциями возникает множество вопросов, например, каким способом передается параметр-функция. Очевидный отрицательный ответ — не по значению. Тело функции не попадает в стек — это совершенно ясно. Тогда вариантов два: по ссылке и по указателю. И опять надо вспомнить, что еще в C были реализованы функции-параметры, когда ссылок не было. А тогда становится совершенно очевидно, что, как и для массивов, имя функции является указателем.

Экономия памяти

Мы знаем, что каждая переменная, объявленная в программе, при трансляции получает собственный адрес. Однако в некоторых задачах бывает удобно одну и ту же область памяти, в разное время работы программы, рассматривать как разные переменные с различной внутренней структурой. Типичным примером является ячейка электронной таблицы — в любой ячейке могут находиться следующие данные:

- строка текста;
- число;
- формула, причем она включает данные двух видов — саму формулу и значение, вычисленное по ней.

Однако в каждый конкретный момент ячейка может быть занята только одним типом данных. Эти данные, конечно, можно объявить как поля структуры, но расход памяти будет слишком велик (вспомните, каких размеров может достигать таблица в Excel). Поэтому в языки C и C++ включена такая конструкция, которая позволяет экономить память как раз в подобных случаях. Эта структура называется `union` — объединение. Синтаксис объявления объединения отличается от синтаксиса объявления структуры только словом `union`:

```
union DD { int a; double b; };
```

Как и структура, конструкция `union` тоже является классом, и в ней можно объявлять функции-методы. В *гл. 3* мы разобрались, сколько памяти занимает структура (см. листинг 3.26). Выполним новый вариант программы, текст которой приведен в листинге 4.20, заменив слово `struct` на слово `union`.

Листинг 4.20. Размеры объединений

```
#include <iostream.h>
int main()
{  union AAA { char ch; double b; };
   union BBB { double b; char ch; };
   union CCC { int b; char ch; };
   union DDD { short b; char ch;
               void f(void) { cout << "Hello!" << endl; }
             };
cout << "AAA" << sizeof(AAA) << endl;
cout << "BBB" << sizeof(BBB) << endl;
cout << "CCC" << sizeof(CCC) << endl;
cout << "DDD" << sizeof(DDD) << endl;
return 0;
}
```

При выполнении данная программа выводит на экран:

```
AAA 8
BBB 8
CCC 4
DDD 2
```

Как видим, методы в объединении, как и в структуре, не занимают место в памяти. В отличие от структуры, при тех же режимах трансляции, под объединение выделяется столько памяти, сколько требуется для наибольшего элемента — все элементы размещаются в памяти по одному адресу. Для объявленного выше объединения `DD` эта ситуация представлена на рис. 4.9.

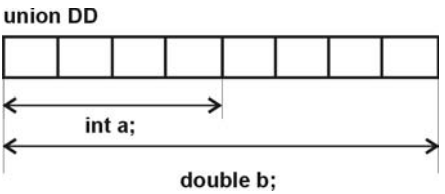

Рис. 4.9. Размещение элементов объединения типа `DD` в памяти

Теперь напишем структуру (ее текст приведен в листинге 4.21) для ячейки электронной таблицы.

Листинг 4.21. Ячейка электронной таблицы

```
struct CELL
{  char attrib;                    //  признак типа информации в ячейке
   union                           //  один из следующих типов
   {  char text[MAX + 1];          //  текст
      double value;                //  число
      struct                       //  формула
      {  double fvalue;            //  значение
         char formula[MAX + 1];    //  строка-выражение
      } f;
   } v;
};
```

Конечно, константа `MAX` должна быть где-то определена. Поле-атрибут, которое является индикатором типа, можно было бы объявить как перечислимый тип, однако и так понятно, что в этом поле будет храниться значение, обозначающее тип данных, с которым в настоящий момент выполняется работа. Доступ к элементам объединения точно такой же, как и к элементам структуры:

```
CELL A;
Strcpy(A.v.text, "Текст в ячейке");
```

Не забывайте, что полю-массиву символов нельзя присвоить значение-строку операцией присваивания, поэтому мы использовали функцию копирования строки.

Программист должен помнить, что в каждый конкретный момент времени "работает" только одно из полей. Когда в ячейку помещается формула, то в программе могут выполняться следующие операторы:

```
Strcpy(A.v.f.formula, "sin(a1) * 0.12");
A.v.f.fvalue = Calculate();
```

Параметры функции *main*

С появлением операционной системы Windows принято различать *консольные* и *оконные приложения*. Консольные приложения запускаются из командной строки. Язык C создавался достаточно давно, поэтому был ориентирован на создание консольных приложений. C++ унаследовал эту черту. Поэтому в данном разделе описывается главная функция для консольных приложений, поскольку она входит в стандарт C++.

Выполнение консольного приложения, написанного на языке C++, всегда начинается с выполнения функции `main`. Эта функция тоже может принимать и обрабатывать параметры, но это будут параметры командной строки. Для того чтобы в функции `main` можно было обрабатывать параметры командной строки, необходимо в заголовке указать два аргумента `argc` и `argv` следующим образом:

```
int main(int argc, char *argv[])
```

Второй допустимой формой заголовка главной функции является следующая запись:

```
int main(int argc, char **argv)
```

Вообще говоря, имена параметров командной строки могут быть любыми, но их принято называть `argc` (argument count — счетчик аргументов) и `argv` (argument vector — вектор аргументов) — в стандарте C++ использованы именно эти имена. Параметр `argc` содержит количество параметров командной строки. Этот параметр всегда не меньше 1, т. к. имя программы считается первым (вернее, нулевым) параметром командной строки. Параметр `argv` — это указатель на массив указателей на строки, каждая из которых является отдельным параметром. Если мы в командной строке задали команду

```
> echo Привет от аргументов
```

то состояние `argv` будет таким, как показано на рис. 4.10.

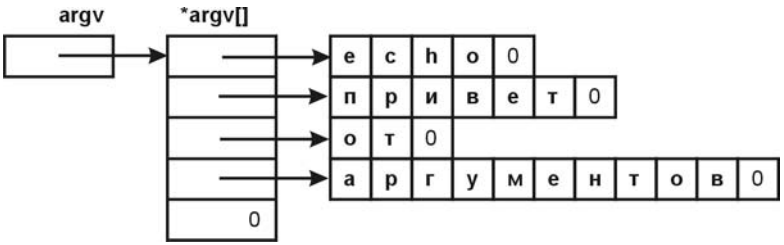

Рис. 4.10. Параметры командной строки

Стандарт C++ требует, чтобы `argv[argc]` всегда был нулевым указателем.

Вообще говоря, в разных операционных системах правила написания параметров командной строки разные. Как принято в Windows (по наследству от MSDOS), параметры в командной строке разделяются пробелами (или символом табуляции). Если нужно, чтобы параметр в командной строке содержал пробелы, надо его заключить в кавычки "". Если аргументом командной строки является число, то оно тоже передается в главную функцию как строка и должно быть преобразовано с использованием соответствующих функций преобразования, вроде `atof()` и др.

Рассмотрим простейшую программу (ее текст приведен в листинге 4.22), которая получает один параметр командной строки и выводит его на экран. Программа корректно транслируется и выполняется в любой системе.

Листинг 4.22. Параметры функции `main`

```
#include <iostream.h>
int main(int argc, char *argv[])
{  if (argc != 2)    //  проверяем количество параметров
   {  cout << "Вы забыли задать аргумент в командной строке!" << endl;
      return 1;
   }
   cout << "Здравствуйте!" << argv[1] << endl;
// if (argv[argc] == 0) cout << "Стандарт поддерживается!" << endl;
   return 0;
}
```

Если выполняемый файл программы имеет имя VVV.EXE, то мы можем его запускать из командной строки, например,

```
> VVV "Удивительно теплая погода сегодня, не так ли?"
```

Программа выдаст на экран сообщение:

```
Здравствуйте! Удивительно теплая погода сегодня, не так ли?
```

Если ничего не указать при вызове

```
> VVV
```

то на экран будет выдано предупреждение:

```
Вы забыли задать аргумент в командной строке!
```

Аргументы главной функции `argc` и `argv` включены и в стандарт языка C (C99), и в стандарт C++, поэтому реализованы во всех без исключения компиляторах. Однако многие компиляторы реализуют различные расширения, зависящие от операционной системы и конкретной реализации транслятора. В справке Borland C++ Builder 6 написано, что заголовок главной функции может иметь следующие формы:

```
int main()
int main(int argc)
int main(int argc, char * argv[])
int main(int argc, char * argv[], char * env[])
```

Те же формы заголовка допускает и Visual C++ 6. Третий параметр обеспечивает доступ к переменным среды. Переменной среды является, например, `path`. Имя параметра может быть любым, однако по традиции принято именовать третий параметр как `env`, происходящий от слова environment.

Функция `main` должна — по определению в стандарте языка — возвращать значение типа `int`. Делается это, как и во всех функциях, определяемых программистом, с помощью оператора `return`. Возвращаемое значение передается в вызывающий процесс. В наших программах это всегда операционная система (командный процессор). Возвращаемое значение обычно можно проверить командой операционной системы.

Если функция `main` не использует параметры командной строки и не возвращает код возврата, то она может быть объявлена следующим образом:

```
void main(void) { //... }
```

Те компиляторы, на которых мне приходилось работать, правильно обрабатывают такую ситуацию. Однако, как указано в [45], это является "ересью", поскольку в стандарте языка явно указано, что главная функция обязана возвращать значение типа `int`.

ГЛАВА 5

Стандартная библиотека

До 1998 года стандартной библиотеки C++ официально не существовало. Однако "святая простота" массивов и отсутствие полноценных строк в языке серьезно осложняли жизнь программистам. Поэтому естественным следствием такого "безобразия" стало то, что каждая система программирования для C++ предлагала свои собственные библиотеки. Этот "разнобой" приводил к тому, что несмотря на провозглашенный принцип о переносимости программ на C++, перейти на другую систему программирования было достаточно сложно даже в одной операционной системе. Поэтому программисты часто писали свои собственные реализации так необходимых строк и других нужных структур данных. И вот в 1994 году Александр (Алекс) Степанов (совместно с Meng Lee) предложил комитету по стандартизации библиотеку контейнеров, которую с того времени обычно называют *STL* (Standart Template Library — Стандартная библиотека шаблонов). Эта библиотека и была практически полностью включена в стандарт C++. Стандарт постоянно уточняется и корректируется, о чем сообщается в дополнительных выпусках-релизах — сначала неофициальных, а затем, после соответствующей доработки, в официальных.

Надо сказать, что разработчики трансляторов просто не поспевают за членами комитета по стандартизации, поэтому часто оказывается, что некоторое свойство работает не совсем так, как прописано в стандарте. Кроме того, и ошибок в системах бывает достаточно много. В такой ситуации остается только следить за изменениями и искать в Интернете более "свежую" стандартную библиотеку. Новую библиотеку можно "прикрутить" к вашей системе вместо той, которая входит в поставку вашей системы (например, Visual C++ 6). Кстати, бесплатные версии, как правило, выставляются с исходными текстами, поэтому можно "покопаться" в недрах STL, что существенно повысит (или не повысит) вашу квалификацию.

Логический тип данных

В стандарт включен логический булевский тип данных, который называется `bool`, теперь `true` и `false` являются зарезервированными словами C++ и их нельзя применять в другом смысле. Новые операции вводить не потребовалось, т. к. они были определены еще в C — это известные нам логическое И (`&&`), логическое ИЛИ (`||`) и логическое НЕ (`!`).

Булевский тип данных можно использовать так же, как и другие встроенные типы, в частности, передавать в функции как по значению, так и по ссылке, возвращать в качестве результата, объявлять массивы и включать в качестве полей в структуры. Размер `sizeof(bool)` равен `1`. Однако вы помните, что выражения с операциями сравнения *(см. разд. "Операции присваивания и выражения" гл. 1)* по умолчанию преобразовывались в целые `0` или `1` в зависимости от результата. Теперь допускается и обратное преобразование: ноль соответствует булевскому значению `false`, а не ноль — `true`. Более того, теперь мы можем и на экране получить значение булевского типа в естественном виде. Для этого достаточно применить новый манипулятор `boolalpha`, например:

```
cout << boolalpha << (a > 1) << endl;
```

На экран будет выведено `true` или `false` в зависимости от результата операции сравнения.

Замечание

Системы Borland C++ 3.1 и Borland C++ 5 "не понимают" этого манипулятора, поскольку реализованы раньше принятия окончательного варианта стандарта.

Новые строки

Массивы символов доставляли много неприятностей программистам, поэтому в стандартную библиотеку C++ был включен полноценный класс строк `string`. Хотя "стандартный библиотечный класс `string` не идеален" [37, с. 645], он предоставляет программисту множество операций для манипулирования строками: разнообразнейшую инициализацию при объявлении; присваивание, добавление и сцепление. Серьезная обработка строк требует наличия операций сравнения, поиска, вставки, замены и удаления подстрок, обязательно должен быть реализован ввод и вывод строк. Большинство этих операций реализованы и как методы класса `string`, и как стандартные алгоритмы, которые мы будем постепенно изучать. Доступ к отдельному символу строки делается по индексу операцией `[]`. Кроме того, реализован последовательный доступ с помощью *итераторов*.

Тип `string` неплохо взаимодействует с символьными массивами: мы всегда можем переписать символы из массива в строку и наоборот; длину строки

тоже всегда можно определить. Естественно, т. к. `string` — это класс, то большинство операций либо перегружены, либо реализованы как методы. Большинство способов инициализации и ряд простых операций продемонстрированы в программе, текст которой приведен в листинге 5.1.

Листинг 5.1. Простые операции со строками

```
#include <iostream>
#include <string>
using namespace std;
int main(void)
{  string s0;                                  //  00
   string s1 = "11111111111111111111"          //  01
   string s3("98765432109876543210");          //  02
   char cs4[] = "01234567890123456789";        //  03
   string s2(s1);
   cout << "s2 =" << s2 << endl;               //  04
   string s4(cs4);
   cout << "s4 =" << s4 << endl;               //  05
   string s5(cs4, 15);
   cout << "s5 =" << s5 << endl;               //  06
   string s6(cs4, 15, 5);
   cout << "s6 =" << s6 << endl;               //  07
   string s61(cs4, cs4 + 4);
   cout << "s61 =" << s61 << endl;             //  08
   //string s7(s3, 15);
   cout << s7 << endl;                         //  09
   string s8(s3, 15, 5);
   cout << "s8 =" << s8 << endl;               //  10
   string s7(s3.begin(), s3.end());
   cout << "s7 ="<< s7 << endl;                //  11
   string s71(s3.begin(), s3.begin() + 6);
   cout << "s71 =" << s71 << endl;             //  12
   string s72(s3.begin(), 15);
   cout << "s72 ="<< s72 << endl;              //  13
   string s73(s3.end() - 7, s3.end());
   cout<< "s73 =" << s73 << endl;
   string s74(s3.begin() + 6, s3.end() - 7);
   cout << "s74=" << s74 << endl;
   string s9(20, '-');
   cout << "s9 =" << s9 << endl;               //  14
   s1 = s3;
   cout << "s1 =" << s1 << endl;               //  15
   s2 += s1;
```

```
   cout << "s2 =" << s2 << endl;           //  16
   s3 = s3 + s9;
   cout << "s3 =" << s3 << endl;           //  17
   s4 += "abcdef";
   cout << "s4 =" << s4 << endl;           //  18
   s4 += '1';
   cout <<"s4 =" << s4 << endl;           //  19
   s4 = s4 + "abcdef";
   cout << "s4 =" << s4 << endl;           //  20
   s4 = s4 + '2';
   cout << "s4 =" << s4 << endl;           //  21
   s5 = s1[0] + s9;
   cout << "s5 =" << s5 << endl;           //  22
   cout << "s7.begin() =" << s7.begin() << endl;            //  23
   cout << "s7.begin()[2] =" << s7.begin()[2] << endl;      //  24
   cout << "s7.begin() + 2 =" << s7.begin() + 2 << endl;    //  25
   cout << "s1.length =" << s1.length() << endl;            //  26
   cout << "s1.size =" << s1.size() << endl;                //  27
   int n;
   cin >> n;
   string aa(n, '-');                                      //  28
   return 0;
}
```

Сначала рассмотрим варианты объявления и операции, а потом разберем "шапку" программы. Переменные `s0`, `s1`, `s2`, `s3` демонстрируют "обыкновенные" способы объявления и инициализации. Переменные `s4`, `s5`, `s6`, `s61` показывают, как для инициализации может использоваться символьный массив. В `s4` скопирован весь символьный массив, в `s5` попали только первые 15 символов этого массива, а в `s6` — 5 символов, начиная с 15-го. Инициализация переменной `s6` в строке `08` показывает наиболее часто используемую форму задания последовательности. Как видим, это соответствует указательной форме задания параметра-массива.

Строка `09` вызывает ошибку трансляции в Visual C++ 6, хотя системы Borland C++ 5 в режиме расширений и Borland C++ Builder 6 отрабатывают совершенно нормально. В [37, с. 652] приведен аналогичный пример.

Замечание

При использовании с Visual C++ 6 другой версии стандартной библиотеки (например, STLport 4.5.3) эта строка никаких проблем не вызывает.

Переменная `s8` (строка `10`) инициализируется аналогично переменной `s6` (строка `07`): 5 символов строки `s3`, начиная с 15-го. Наконец, переменная `s9` заполняется двадцатью минусами.

Особо надо остановиться на строках программы, в которых объявляются переменные s7, s71, s72, s73, s74 — здесь продемонстрировано простейшее использование итераторов. *Итератор* — это объект стандартной библиотеки, обеспечивающий доступ к элементам контейнера. Строка является контейнером символов (хотя по отношению к строкам и не принято применять термин "контейнер"), поэтому к элементам строки можно обращаться с помощью итераторов. Каждый контейнер имеет два метода: begin() и end(). Названия методов говорят сами за себя, поэтому переменная s7 получает всю строку s3, в переменную s71 помещаются только первые 6 символов из строки s3, а объявление переменной s72 демонстрирует копирование первых 15-ти символов из строки s3. Это — аналог не работающего способа инициализации в строке 09, который мы закомментировали. Переменная s73 получает последние 7 символов строки s3, а в переменную s74 попадают 7 символов той же строки, начиная с 7-го, т. е. s74 равна "3210987". Эти объявления, а также операторы вывода в строках 23, 24, 25 показывают, что работа с итераторами очень похожа на работу с параметрами-массивами в указательной форме *(см. разд. "Параметры-массивы в форме указателя" гл. 4)*. Это на самом деле так, поскольку частным случаем итератора как раз является указатель. Фундаментальное отличие указателей от итераторов заключается в том, что итераторы реализованы посредством других конструкций C++, а указатели являются встроенными (как и элементарные встроенные типы).

Строки программы 15-22 демонстрируют различные варианты операции присваивания и сцепления. Как видим, строки хорошо совместимы со строковыми и символьными константами. Помимо непосредственной операции присваивания в классе string реализован метод assign, который имеет все те же формы, что и инициализация. Если у нас определены две строки A и B и символьный массив C, то можно присвоить значения строке A таким образом:

```
A.assign(B);
A.assign(B, 15, 5);
A.assign(B.begin(), B.end());
A.assign(C);
A.assign(C + 10);
A.assign(100, '+');
```

Нет необходимости приводить остальные формы использования метода assign, поскольку они в точности соответствуют формам инициализации.

И наконец, в строке 26 показано, как узнать длину строки, а в 27 — размер строки. Для контейнеров это может быть не одно и то же.

Строка 28 демонстрирует то, что длина строки может задаваться с помощью переменной, причем строка "забивается" минусами. Мы не могли объявить таким способом символьный массив переменной длины.

"Шапка" программы несколько отличается от той, что мы писали раньше. Во-первых, в операторе `#include` имена включаемых файлов написаны без расширения h. Писать в таком стиле рекомендует стандарт C++. Для совместимости стандарт разрешает писать и в "старом добром" стиле C, однако здесь нас поджидает очередной сюрприз. Если мы напишем string.h, то тем самым затребуем библиотеку для работы с массивами символов. Таким образом, налицо "конфликт поколений" — стандартов C и C++. Чтобы такого впредь не происходило, в стандарте C++ предписано включаемые файлы библиотек C начинать с символа "c". Таким образом, в C++ string.h переименован в cstring. Соответственно ctype.h надо писать как cctype, а math.h — как cmath, и т. д.

Включаемые файлы стандартной библиотеки шаблонов нужно писать по-новому — без расширения. Поэтому мы написали "чистое" string без расширения h. Впредь вместо слов "включаемый файл" будем употреблять термин "заголовок", что звучит более корректно.

Во-вторых, мы написали новый оператор:

```
using namespace std;
```

Без него в системе Visual C++ 6 пример просто не транслируется, выдавая не совсем адекватные сообщения об ошибках. Этот оператор в стандарте называется директивой. Эту директиву желательно писать каждый раз, когда мы используем какие-нибудь средства из стандартной библиотеки. Дословно директива переводится как "используй пространство имен `std`". Пространства имен — это сравнительно новый механизм разрешения конфликтов имен в многомодульных программах. Все имена, так или иначе объявленные в стандартной библиотеке, составляют *стандартное пространство имен*.

Мы могли бы и не писать этой строки, но тогда для корректной работы программы необходимо каждое имя стандартной библиотеки писать с префиксом `std::`

```
std::cout
```

или

```
std::string
```

Согласитесь — это не всегда удобно.

Если вам кажется не совсем правильным подключать полное пространство имен, когда нужно всего одно-два имени, то директива `using` может быть использована для указания и конкретного имени, например:

```
using std::cout;
using std::string;
```

После таких директив можно использовать заданные имена без префикса.

Строки как параметры

Хотя этот тип не является таким простым, как встроенные, передача переменных этого типа в качестве параметров синтаксически и семантически ничем не отличается от передачи переменных встроенных типов: параметры типа `string` можно передавать в функцию по значению, по ссылке и по указателю. Разрешается и возвращать значения типа `string` как результат. Пример использования типа `string` приведен в листинге 5.2.

Листинг 5.2. Использование типа `string`

```
#include <string>                //  не string.h
#include <iostream>
using namespace std;             //  требуется обязательно
string ss(string s)              //  передача по значению
{  s[0] = 'v';                   //  изменения только внутри функции
   return s;                     //  возврат значения
}
string sss(string &s)            //  передача по ссылке
{ s[0] = 'v'; return s; }
int main(void)
{  string s1 ="11111", s2, s3;
   s2 = ss(s1);                        //  изменение s1 не происходит
   cout << s1 << ' ' << s2 << endl;    //  выводит 11111 v1111
   s2 = sss(s1);                       //  изменение s1 происходит
   cout << s1 << ' ' << s2 << endl;    //  выводит v1111 v1111
   return 0;
}
```

Как и для встроенных типов, при передаче по ссылке изменения внутри функции отражаются на переменной вне функции. Как обычно, можно запретить изменения параметра, прописав слово `const` при объявлении функции. Возврат из функции также не отличается от способов возврата элементарных типов: функция может возвращать значение типа `string` или ссылку на тип `string`, — пока мы не будем на этом подробно останавливаться.

Мы можем присвоить параметру-строке значение по умолчанию. Текст примера приведен в листинге 5.3.

Листинг 5.3. Параметр-строка по умолчанию

```
string hh(string s = "По умолчанию")
{ cout << s << endl; return s; }
```

Это делается совершенно аналогично символьным массивам-параметрам *(см. разд. "Указатели на символы" гл. 4)*. Мы не будем более подробно останавливаться на этом вопросе, поскольку варианты присвоения значения по умолчанию ничем не отличаются от рассмотренных ранее для встроенных типов *(см. разд. "Параметры по умолчанию" гл. 2)*.

Так как часто приходится инициализировать строками-константами переменные типа `string`, необходимо написать функцию перекодировки (см. листинг 4.9) и для `string`. Однако здесь нас поджидает один "подводный камень" — функция `CharToOem` не умеет обрабатывать строку типа `string`. Значит, наша задача — "вытащить" из стандартной строки соответствующий массив символов. Естественно, разработчики стандарта предусмотрели такой вариант, поэтому в классе `string` реализован метод `c_str()`, который и позволяет это сделать. Текст примера перекодировки приведен в листинге 5.4.

Листинг 5.4. Перекодировка `string`

```
char * Rus(const string &in, char out[])
{ if (CharToOem(in.c_str(), out)) return out; else return 0; }
```

Обращение к такой функции выглядит точно так же, как и в предыдущем варианте:

```
string ss("Снова привет! Большой и горячий! Hello!");
cout << Rus(ss, s) << endl;
```

Давайте сразу решим и еще одну очень важную проблему. Очевидно, нам практически в каждой программе потребуется выводить на экран сообщение на русском языке. Каждый раз писать заново функцию, и даже просто копировать ее из другого текста — неудобно. Давайте поместим эти две функции в отдельный файл с именем Rus.h и будем просто подключать его с помощью оператора `#include` аналогично стандартным. Только надо этот файл записать в стандартный каталог include. Текст файла Rus.h приведен в листинге 5.5.

Листинг 5.5. Объединенный включаемый файл Rus.h

```
#include <windows.h>
#include <string>
using namespace std;
char ss[100];    //  глобальный массив перекодировки
char * Rus(const char in[], char out[])
{ if (CharToOem(in, out)) return out; else return 0; }
char * Rus(const string &in, char out[])
{ if (CharToOem(in.c_str(), out)) return out; else return 0; }
```

Вначале мы прописали подключение стандартного заголовка windows.h и string. Кроме того, сразу написали оператор использования стандартного пространства имен. Теперь в любой программе, в которой потребуется перекодировать русские строки или просто работать со строками, достаточно написать один оператор:

```
#include <Rus.h>
```

Конечно, все наши вспомогательные файлы писать в стандартный каталог не следует, но в данном случае это приемлемое решение. Далее мы научимся подключать файлы не только из стандартного каталога.

Числа — прописью

Напишем программу, которая чрезвычайно востребована в реальной жизни: вывод числа прописью. Мы не будем писать совсем уж универсальный вариант, т. к. наша цель — просто поучиться использовать тип `string` в реальной программе. Ограничим наши числа положительными целыми в диапазоне от 1 до 999 999 999 — такие числа целиком помещаются в целую переменную типа `int`.

Замечание

Если требуются большие числа, то в системе Visual C++ 6 мы можем использовать нестандартный целый тип `__int64`. Стандартное решение — использовать структуру с двумя полями типа `long`.

Мы не зря написали число 999 999 999 с пробелами между тройками чисел — числительные от 1 до 999 пишутся одинаково, независимо от местоположения. Вся разница — в слове после очередной тройки: старшая — это миллионы, средняя — это тысячи, а младшая — непосредственно рубли. Поэтому естественно написать функцию, которая получает число от 1 до 999 и выдает результат типа `string`. В этом случае мы сможем инкапсулировать все массивы числительных в этой процедуре. Однако миллионы и рубли у нас мужского рода, а тысячи — женского. Поэтому вместо "один" и "два" надо писать "одна" и "две". В остальном числительные полностью совпадают. Поэтому мы должны прописать второй параметр типа `bool`, который всегда будет равен `false`, а для тысяч — `true`. Текст функции для записи трехзначного числа прописью приведен в листинге 5.6.

Листинг 5.6. Функция `toNumeral` — трехзначное число прописью

```
    //  обрабатывает только трехзначное число
string toNumeral(unsigned long Number, bool Thousands)
{ string units[] =
{ "один", "два", "три", "четыре", "пять",
  "шесть", "семь", "восемь", "девять"
};
```

```
string tens[] =
{ "десять", "двадцать", "тридцать", "сорок", "пятьдесят",
  "шестьдесят", "семьдесят", "восемьдесят", "девяносто"
};
string hundreds[] =
{ "сто", "двести", "триста", "четыреста", "пятьсот",
  "шестьсот", "семьсот", "восемьсот", "девятьсот"
};
string secondten[] =
{ "одиннадцать", "двенадцать", "тринадцать", "четырнадцать", пятнадцать",
  "шестнадцать", "семнадцать", "восемнадцать", "девятнадцать"
};
string Women[2] = { "одна", "две" };
string result("");              //  пустая строка-результат
typedef unsigned char byte;
byte digits[3] = { 0 };         //  цифры числа
unsigned long n = Number;
digits[0] = n % 10;             //  младшая цифра
digits[1] = (n / 10) % 10;      //  средняя цифра
digits[2] = n / 100;            //  старшая цифра
if (digits[2] > 0) result += (hundreds[digits[2] - 1] + " ");
if (digits[1] > 0)
{  if ((digits[1] == 1) && (digits[0] != 0))
   { result += (secondten[digits[0] - 1] + " "); return result; }
   else result += (tens[digits[1] - 1] + " ");
}
if (digits[0] > 0)
{  if (((digits[0] > 2) && (digits[1] != 1)) || (!Thousands))
   result += (units[digits[0] - 1] + " ");
   else result += (Women[digits[0] - 1] + " ");
}
return result;
}
```

Вначале объявляются и инициализируются массивы числительных: в каждом массиве ровно 9 элементов. Затем объявляется и заполняется массив для трех цифр числа: цифра сотен помещается в `digits[2]`, цифра десятков — в `digits[1]` и цифра единиц — в `digits[0]`. Наконец, начинаются проверки цифр и получение строки-результата. Первоначально строка-результат пустая. Если очередная цифра не равна нулю, то к строке-результату прицепляется очередное числительное. Особый случай составляет вторая цифра (десятки): если она равна `1`, то мы тут же проверяем третью. Если она не равна нулю, то мы прицепляем числительное из массива `secondten` и выходим. Если цифра десятков не равна `1`, то к строке-результату прицеп-

ляется соответствующее числительное из массива `tens`. При определении числительного единиц для цифр 1 или 2 используется массив `Women` числительных женского рода.

Обратите также внимание на то, как вычисляется индекс числительного — это довольно распространенный прием. Протестируем нашу функцию (текст теста приведен в листинге 5.7).

Листинг 5.7. Тестовая программа для функции `toNumeral`

```
int main()
{  unsigned long Number[] =
  { 1, 2, 9, 10, 11, 19, 20, 21, 22, 45, 100,
    101, 102, 109, 110, 118, 150, 870, 912, 999
  };
int k = sizeof(Number) / sizeof(unsigned long);
char s[200] = { 0 };
for (int i = 0; i < k; ++i)
{  CharToOem(toNumeral(Number[i], true).c_str(), s);
   cout << s<< endl;
}
return 0;
}
```

В этой программе мы объявили тестовый массив `Number` и вычислили его длину. Затем в цикле выполняется непосредственное обращение к функции перекодировки. Мы не стали использовать написанные нами ранее функции `Rus` (см. листинг 4.9, 5.4). Первый операнд написан несколько необычно. Поскольку мы не записали результат в переменную, система создает *анонимный временный объект*. Наша функция `toNumeral` возвращает `string`, поэтому у результата (временного объекта) есть метод `c_str()`. Использование временных объектов — полезный прием, которым мы еще не раз будем применять.

Теперь напишем функцию (ее текст приведен в листинге 5.8), которая будет формировать наше число прописью. Очевидно, эта функция должна трижды вызвать функцию `toNumeral`, "прицепляя" после каждого вызова соответствующее слово.

Листинг 5.8. Сумма — прописью

```
string Numeral(unsigned int Number)
{  string what_[3][3] = { { "миллион", "миллиона", "миллионов" },
                          { "тысяча", "тысячи", "тысяч" },
                          { "рубль", "рубля", "рублей" } };
```

```
unsigned int t = Number;
unsigned int treads[3];    //  вычисление троек числа
treads[0] = t % 1000;
treads[1] = t / 1000 % 1000;
treads[2] = t / 1000000;
string result = "";
if (treads[2] > 0) { result += toNumeral(treads[2], false);
if (treads[2] / 10 % 10 == 1) result += what_[0][2];    //  1
else
switch (treads[2] % 10)
{  case 0: case 5: case 6: case 7: case 8: case 9:
   result += what_[0][2]; break;
   case 2: case 3: case 4: result += what_[0][1]; break;
   case 1: result += what_[0][0]; break;
}
result += ' ';
}
if (treads[1] > 0) {result += toNumeral(treads[1], true);
if (treads[1] / 10 % 10 == 1) result += what_[1][2];    //  2
else switch (treads[1] % 10)
{  case 0: case 5: case 6:
   case 7: case 8: case 9: result += what_[1][2]; break;
   case 2: case 3: case 4: result += what_[1][1]; break;
   case 1: result += what_[1][0];break;
}
result += ' ';
}
result += toNumeral(treads[0], false);
if (treads[0] / 10 % 10 == 1) result += what_[2][2];    //  3
else switch (treads[0] % 10)
{  case 0: case 5: case 6:
   case 7: case 8: case 9: result += what_[2][2]; break;
   case 2: case 3: case 4: result += what_[2][1]; break;
   case 1: result += what_[2][0]; break;
}
result += ' ';
return result;
}
```

Как видите, получилась довольно простая программа. Единственное, на что надо обратить внимание, — это помеченные цифрами строки: если у нас вторая цифра в тройке равна `1`, то можно сразу прицеплять нужное слово. В остальных случаях мы должны анализировать последнюю цифру. Можно еще больше упростить функцию, если выделить в отдельную функцию со-

вершенно однотипные действия в операторе `switch`, но оставим это на упражнения.

Протестировать нашу программу очень просто: надо в листинге 5.7 увеличить тестовый массив, задав уже и тысячи, и миллионы, и заменить первый параметр в вызове функции `CharToOem`:

```
CharToOem(Numeral(Number[i]).c_str(), s);
```

Контейнеры, итераторы и алгоритмы

Основу стандартной библиотеки составляют контейнеры, итераторы и алгоритмы. *Контейнер* — это набор однотипных элементов. Как мы помним, массив — это тоже набор однотипных элементов, однако он недостаточно "умный". Контейнеры, в отличие от массивов, "знают" свой размер и могут его изменять в процессе работы. Кроме того, для массивов *(см. гл. 3)* нам пришлось писать довольно много функций обработки. В стандартной библиотеке уже есть более 60 типовых алгоритмов для обработки контейнеров. При этом алгоритмы написаны таким образом, что без особых проблем их можно использовать и для массивов, которые с точки зрения алгоритмов тоже считаются контейнерами.

Контейнеры обладают собственными методами, поэтому любой стандартный контейнер можно обрабатывать либо с помощью стандартного алгоритма, либо применяя методы самого контейнера. Строки `string` тоже являются наборами однотипных элементов, хотя и не считаются контейнером. Тем не менее и к строкам разрешается применять многие алгоритмы. Строки имеют многие методы, которые называются так же, как методы контейнеров, и работают аналогично.

Все контейнеры делятся на два класса: *последовательные* и *ассоциативные*. Кроме того, обычно различают еще собственно контейнеры и *адаптеры контейнеров*. Мы этого делать не будем, поскольку использование тех и других ничем друг от друга не отличается. Итак, к последовательным контейнерам относятся *вектор* (`vector`) и *список* (`list`), *очередь* (`queue`) и *приоритетная очередь* (`priority_queue`), *дек* (`deque`) и *стек* (`stack`). С точки зрения их использования в программе эти контейнеры отличаются только *дисциплиной доступа к элементам*. В зависимости от дисциплины доступа контейнер содержит те или иные методы (табл. 5.1). Но следует заметить, что многие методы унифицированы и совпадают как по прототипу, так и по результату работы. Кроме того, использование итераторов обеспечивает универсальный механизм доступа — последовательный.

К ассоциативным контейнерам относятся *множества* (`set`) и *мультимножества* (`multiset` — множества с повторяющимися элементами), *словари* (`map`) и *мультисловари* (`multimap` — словари с повторяющимися элементами).

Примечание

В системе Borland C++ Builder 6 версия STL (STLport 4.5.3) включает еще несколько контейнеров, например `slist` или `hash_map`.

Таблица 5.1. *Операции доступа к элементам последовательных контейнеров*

Операция	Метод	vector	deque	list	stack	queue
Вставка вперед	push_front	–	+	+	–	–
Удаление первого	pop_front	–	+	+	–	+
Вставка в конце	push_back	+	+	+	+	+
Удаление последнего	pop_back	+	+	+	+	–
Вставка в любое место	insert	+	+	+	–	–
Удаление произвольное	erase	+	+	+	–	–
Произвольный доступ	[ ], at	+	+	–	–	–

Словари часто называют отображениями. Хотя множества сами по себе — это не ассоциативная структура, множества стандартной библиотеки отнесли к ассоциативным контейнерам, подчеркнув специфику реализации. Основная идея ассоциативных контейнеров состоит в том, что доступ к элементам осуществляется по ключу. Проще всего объяснить это на примере массива. Пусть у нас есть некоторый гипотетический массив `W`, в котором в качестве индекса используется имя клиента, а значением элемента является его банковский счет. Тогда мы могли бы вывести на экран информацию о счете клиента, написав такой оператор:

```
cout << W["Вася"] << endl;
```

Строка "Вася" — это ключ, а наш массив `W` — ассоциативный. В C++ с массивами такое делать нельзя, а в стандартной библиотеке именно для этого и реализованы ассоциативные контейнеры.

Векторы вместо массивов

Опять обратимся к проблеме среднего арифметического (см. листинги 1.20 и 1.21). В *гл. 2* (см. листинг 2.13) мы использовали статические переменные для накопления суммы и количества, поскольку нам заранее было неизвестно, какое количество чисел задаст пользователь. Теперь мы можем об этом не думать, т. к. в стандартной библиотеке шаблонов реализованы *динамические контейнеры*. Слово "динамические" означает, что размер объявленного контейнера может изменяться по мере обработки. По крайней

мере, за увеличением количества элементов мы можем не следить, пока хватает памяти компьютера.

Для ввода неизвестного количества чисел нам подходят все последовательные контейнеры. Однако традиционно (и стандарт рекомендует) вместо массива применяется вектор. Для того чтобы использовать любой контейнер, мы должны прописать в программе операторы:

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

Вектор так и называется `vector`. Объявлять, инициализировать и присваивать вектор можно следующим образом (текст примера приведен в листинге 5.9).

Листинг 5.9. Объявление и инициализация векторов

```
vector <double> u;                      //  1 — пустой вектор
vector <int> v(10);                     //  2
vector <int> x(15, -1);                 //  3
vector <int> t(v.begin(), v.end());     //  4
int a[10] = { 1, 2, 3, 4, 5 };
vector <int> w(a, a + 10);              //  5
int n;
cout << "Input n: ";
cin >> n;
vector <int> vv(n, 1);                   //  6
tt = w;                                  //  7 — размер tt = размер w
for( i = 0; i < tt.size(); ++i) cout << tt[i] << ' ';    //  8
cout << endl;
t.assign(20, -1);
```

Поскольку вектор — это шаблон, то после слова `vector` надо, в соответствии с правилами объявления шаблонов *(см. разд. "Шаблоны структур" гл. 3)*, прописать тип элементов в угловых скобках. Этот тип может быть любым из допустимых: как встроенным, так и одним из реализованных типов.

В строке `1` объявляется "пустой" вектор, не содержащий ни одного элемента типа `double`. Элементы могут быть добавлены к вектору с помощью метода `push_back`. Естественно, "пустоту" вектора всегда можно определить с помощью метода `empty()`, который выдает `true`, если вектор не содержит ни одного элемента.

В строке `2` объявлен вектор из 10-ти чисел типа `int`, причем по умолчанию все числа равны нулю.

Строка `3` демонстрирует объявление вектора из 15-ти целых чисел, причем все элементы вектора равны `-1`.

В строке 4 объявлен вектор, который инициализируется последовательностью элементов ранее объявленного вектора, причем последовательность задается парой итераторов. В такой форме инициализации допустимы все варианты, которые мы ранее привели для типа string.

Строка 5 демонстрирует инициализацию элементов вектора элементами ранее объявленного массива. В этом случае дозволяются любые выражения, допустимые для параметров в указательной форме.

В строке 6 показано, что в отличие от массива мы можем объявить вектор переменного размера, длина которого задается не константой, а переменной. Хотя вектор и так является динамической структурой, иногда это может пригодиться.

Как видим, формы объявления и инициализации векторов во многом совпадают с объявлениями и инициализацией строк. Это сделано специально, чтобы упростить жизнь программистам. Более того, формы объявления и инициализации других контейнеров тоже практически совпадают с представленными формами для векторов и строк. В общем-то всего реализовано четыре способа объявления и инициализации.

1. Без аргументов.
2. С аргументами-итераторами, в частности, указателями.
3. С количеством элементов и значением по умолчанию.
4. Другим контейнером.

Метод assign перегружен в тех же формах.

Операция присваивания для векторов разрешена, однако принимающий вектор получает не только набор значений из правого вектора, но и его размер. В данном случае (строка 7) размер вектора tt станет равным размеру вектора w. Типы элементов должны совпадать, иначе программа не транслируется.

Далее мы видим, что обращаться к элементу вектора можно точно так же, как и к элементу массива. Однако операция индексирования для вектора не проверяет границы, поэтому мы не застрахованы от ошибки "вылета за границу". Такое решение принято из соображений эффективности. Поэтому принят компромиссный вариант: реализован специальный метод at, который проверяет границы. Если нам важна надежность, то вместо v[i] мы должны писать v.at(i). В условии цикла прописан вызов метода size(), который выдает текущее на данный момент количество элементов вектора.

В последней строке листинга 5.9 продемонстрировано применение метода assign, с помощью которого можно присвоить вектору новые значения и изменить его размер. Метод перегружен и имеет такие же формы, как и уже приведенные варианты инициализации.

А теперь вернемся к задаче вычисления среднего. Алгоритм будет такой: все введенные числа помещаются в вектор, а затем этот вектор передается функции вычисления среднего в качестве параметра. Функция возвращает результат — число. Первая версия программы приведена в листинге 5.10.

Листинг 5.10. Вычисление среднего с вектором, версия 1

```
#include <vector>
using namespace std;
double Average(const vector <double> &v)
{  double s = 0.0;
   for (int i = 0, count = 0; i < v.size(); ++i)
   s += v[i], ++count;    //  один оператор
   return s / count;
}
```

Вектор передается в функцию `Average` по ссылке. В отличие от массивов, которые мы принципиально не можем передать по значению, а только по указателю, способ передачи параметров-контейнеров в функцию целиком зависит от программиста. Допускается передавать вектор в функцию по значению, однако это весьма накладная операция: во-первых, существенно расходуется память, поскольку весь вектор должен быть размещен в стеке; во-вторых, эта же причина влияет на скорость работы программы, т. к. выполняются команды записи элементов вектора в стек. Поэтому всегда передавайте контейнеры по ссылке. Если требуется запретить изменение элементов контейнера, что бывает весьма редко, прописывайте константность параметра.

Проверить нашу функцию можно так:

```
#include <iostream>
#include <Rus.h>       //  подключение русификации
int main()
{  vector <double> u; double el;
   while (cout << Rus("Задайте элемент вектора: ", ss), cin >> el)
   u.push_back(el);    //  "наполнение" вектора
   cout << Average(u) << endl;
   return 0;
}
```

В основной программе вводятся числа и накапливаются в векторе `u`. Нам нет необходимости как-то отслеживать этот процесс — элемент просто присоединяется в конец вектора. Заменив вектор списком или деком, мы получаем возможность присоединять элементы не только в "хвост", но и в начало контейнера:

```
list <double> u;     //  можно и deque
while (cout << Rus("Задайте элемент списка: ", out), cin >> el)
u.push_front(el);    //  "наполнение" списка
```

Функция прекрасно решает поставленную задачу, однако результата можно достичь и без явного программирования цикла, если применить стандартный алгоритм `accumulate`. Тогда наша функция выглядит совсем просто:

```
double Average(const vector <double> &v)
{  if (!v.empty()) return
   (accumulate(v.begin(), v.end(), 0.0) / v.size());
   return 0;
}
```

Чтобы функция никогда не была причиной аварии программы, в которой она используется, мы проверяем пустоту вектора. Для успешной работы необходимо прописать в шапке программы оператор:

```
#include <numeric>
```

Три последовательных контейнера: `vector`, `list` и `deque`, — являются во многих (но отнюдь не во всех [25]) случаях взаимозаменяемыми, поэтому результат работы программы не изменится, если мы заменим `vector` на `list` или `deque`, прописав соответствующий оператор `#include`. Новая версия программы приведена в листинге 5.11.

Листинг 5.11. Вычисление среднего с контейнером `deque`

```
#include <iostream>
#include <deque>
#include <numeric>
using namespace std;
#include <Rus.h>       //  подключение русификации
double Average(const deque <double> &v)
{  if (!v.empty())
   return (accumulate(v.begin(), v.end(), 0.0) / v.size());
   else return 0;
}
int main()
{  deque <double> u; double el;
   char out[100];      //  для перекодированной строки
   while (cout << Rus("Задайте элемент дека: ", out), cin >> el)
   u.push_back(el);    //  "наполнение" дека
   cout << Average(u) << endl;
   return 0;
}
```

Контейнеры можно не только передавать в качестве параметров, но и возвращать как результат. Чтобы продемонстрировать это, напишем функцию, которая получает в качестве параметра контейнер большого размера, а возвращает контейнер, являющийся ограниченной выборкой n случайных элементов этого контейнера. Такая задача может возникнуть, например, при реализации системы тестирования, где из большого общего набора тестовых вопросов надо сгенерировать небольшой конкретный тест из ограниченного количества вопросов. Предположим, что тип элемента контейнера — string. Тогда прототип функции, с учетом использования оператора typedef, выглядит следующим образом:

```
typedef vector <string> Question;
Question GetTest(const Question &Test, int n)
```

Схема реализации функции выглядит так:

```
инициализировать датчик случайных чисел;
Объявить вектор КонкретныйТест;
i = 0;
пока (i < n+1)
   генерировать случайное число m;
   если m не повторяется, то
   включить в КонкретныйТест Вопрос[m];
   ++i;
конец пока
вернуть КонкретныйТест
```

Проверка повторной генерации случайного числа m представляет собой некоторую проблему. Однако мы можем ее решить, объявив множество целых чисел. Как известно, контейнер set отличается тем, что содержит единственный экземпляр каждого элемента.

Сначала множество будет пустым. Каждый раз при генерации числа мы будем проверять наличие этого числа в множестве. Если его там нет, то мы его туда включаем, а соответствующий элемент исходного вектора добавляем к вектору-результату. С учетом этих соображений реализация не представляет особой сложности. Текст примера приведен в листинге 5.12.

Листинг 5.12. Возврат вектора

```
typedef vector <string> Question;
Question GetTest(const Question &Test, int n)
{  time_t t;                        //  стандартная
   srand((unsigned) time(&t));      //  инициализация датчика
   Question ConcreteTest;           //  вектор-результат
   Set <int> s;    //  множество
```

```
   int i = 0;
   while(i < n)
   {  int k = rand() % Test.size();           //  генерация числа
      set <int> :: iterator ik = s.find(k);   //  проверка на повторение
      if (ik == s.end())                      //  если число новое
      {  s.insert(k);                         //  включаем в множество
         ConcreteTest.push_back(Test[k]);     //  присоединяем вопрос
         ++i;
      }
   }
   return ConcreteTest;             //  возврат вектора-результата
}
```

Чтобы функция работала, необходимо подключить заголовок:

```
#include <set>
```

Проверка наличия сгенерированного числа в множестве выполняется методом `find`, который, как обычно, выдает результат-итератор. Если числа в множестве нет, то этот итератор равен `end()`. Пока мы не научились работать с файлами, нам сложно представить вектор строк большого объема. Поэтому проверку нашей функции выполним с помощью вектора, содержащего названия месяцев года, который заполним так:

```
Question vv;
vv.push_back("январь");
vv.push_back("февраль");
vv.push_back("март");
vv.push_back("апрель");
vv.push_back("май");
vv.push_back("июнь");
vv.push_back("июль");
vv.push_back("август");
vv.push_back("сентябрь");
vv.push_back("октябрь");
vv.push_back("ноябрь");
vv.push_back("декабрь");
```

Вызов нашей функции и вывод результатов теперь делаются просто:

```
Question tt = GetTest(vv, 5);
for (int mm = 0; mm < tt.size(); ++mm) cout << Rus(tt[mm], ss) << endl;
```

На экран будет выведено пять из 12 названий месяцев, например:

```
сентябрь
январь
```

```
апрель
февраль
ноябрь
```

Названия выводятся, естественно, не по порядку, т. к. числа генерируются не в порядке возрастания. Если мы хотим выводить строки по алфавиту, то надо просто отсортировать вектор либо прямо в функции, либо в главной программе.

Указатели и контейнеры

Указатели и контейнеры стандартной библиотеки прекрасно уживаются друг с другом. Элементами контейнеров, если это необходимо, могут быть указатели, например:

```
vector <void *> v;
```

Можно объявить указатель на контейнер, а также создать динамическую переменную-контейнер, например:

```
vector <int> *m = new vector <int>;
```

Динамический контейнер, в отличие от динамического массива, можно инициализировать, например, так:

```
typedef vector <double> row;
row *m = new row(10, 2.1);
```

Мы создали динамический вектор из 10 элементов и проинициализировали каждый элемент значением `2.1`. Доступ по индексу тогда делается так:

```
cout << (*m)[i] << endl;
```

или так:

```
cout << m -> at(i) << endl;
```

Доступ к элементам такого вектора может быть выполнен и по итератору. При этом, естественно, используется операция доступа `->`:

```
row *mm = new row(10, 2);            //  объявление и инициализация
mm -> push_back(1);                  //  добавление 1 в конец
cout << mm -> size() << endl;        //  1
cout << mm -> capacity() << endl;    //  2
vector <double> :: iterator k;
for (k = mm -> begin(); k < mm -> end(); ++k) cout << setw(4) << *k;
cout << endl;
```

В строке `1` выводится на экран количество элементов вектора (в данном случае `11`), а в строке `2` — количество зарезервированных элементов вектора

(в системе Visual C++ 6 — `20`). Далее объявляется итератор `k`, который используется для вывода элементов вектора на экран.

Стандартные алгоритмы и итераторы

На примере обработки массивов *(см. гл. 3)* мы знаем, что типовыми операциями при обработке однородных контейнеров являются сортировка и поиск, получение различных сумм из элементов контейнера, операции над каждым элементом контейнера. Все эти алгоритмы реализованы в составе библиотеки STL в максимально общем виде. Один и тот же алгоритм практически всегда (исключения очень редки) может обрабатывать и контейнеры, и массивы.

Параметрами алгоритмов, как правило, является пара итераторов, задающая последовательность элементов контейнера (вспомним функцию вычисления суммы массива, параметрами которой были два указателя — см. листинг 4.1). В простейшем виде мы уже умеем ими пользоваться. Итератор можно рассматривать как некий *объект-посредник*, который обеспечивает клиенту (нашей программе) доступ к элементам контейнера, не раскрывая ни внутренней структуры контейнера, ни структуры элемента контейнера.

Примечание

Итератор — это один из широко распространенных *паттернов проектирования*. Паттерн проектирования — это типовой прием в разработке и реализации программного обеспечения. Иногда паттерн "Итератор" называют именем "Курсор".

Таким образом, итераторы обеспечивают очень высокую степень инкапсуляции. А как показала многолетняя практика программирования, чем выше инкапсуляция, тем проще модифицировать программу в случае необходимости, т. к. существенно облегчается поиск ошибок.

Итераторы имеют тип, связанный с типом контейнера, поэтому для определенности будем работать с контейнером-списком `list` с элементами типа `double`, тем более что список не позволяет обращаться к своим элементам с помощью индекса. Напишем небольшую программу (ее текст приведен в листинге 5.13), в которой продемонстрируем обращение к элементам контейнера с помощью итераторов.

Листинг 5.13. Доступ к элементам контейнера через итератор

```
#include <iostream>
#include <algorithm>
#include <list>
#include <cstdlib>
#include <ctime>
```

```
using namespace std;
#include <Rus.h>
int main()
{  double mu[10];
   time_t t;                                              //  1
   srand((unsigned) time(&t));                            //  2
   for (i = 0; i < 10; ++i) mu[i] = (rand() % 100);       //  3
   list <double> L(mu, mu + 10);                          //  4
   list <double> :: iterator il;                          //  5
   list <double> :: reverse_iterator ril;                 //  6
   for (il = L.begin(); il != L.end(); ++il) cout << *il << ' ';
   cout << endl;
   for (ril = L.rbegin(); ril != L.rend(); ++ril) cout << *ril << ' ';
   cout << endl;
   return 0;
}
```

Эта программа выдаст на экран две строки:

```
77 59 24 23 3 6 97 45 10 96
96 10 45 97 6 3 23 24 59 77
```

представляющие собой содержимое одного и того же списка `L`. Мы не изменяли сам список, просто вывели его сначала в прямом, а потом в обратном порядке. Сначала в программе объявлен массив `mu`, который заполняется с помощью стандартного датчика случайных чисел. Датчик, так же как и в листинге 5.12, инициализируется функцией `srand`, аргументом которой является вызов функции `time` (заголовок — `ctime`). Эта функция возвращает количество секунд от начала отсчета 1 января 1970 года, принятого на IBM PC. Очередное случайное число выдается функцией `rand`. Прототипы `srand` и `rand` прописаны в cstdlib.

В строке `4` объявлен и проинициализирован список `L`, а в следующих двух строках объявляются итераторы: *прямой* (строка `5`) и *обратный* (строка `6`). Прямой итератор позволяет "пробежать" элементы от начала к концу контейнера, а обратный — от конца контейнера к началу. Это разные типы итераторов и смешивать их нельзя.

В первом цикле прямой итератор инициализируется уже известным нам способом — с помощью метода `begin()`. Во втором цикле для инициализации используется совсем другой метод — `rbegin()`. Соответственно для окончания цикла вызывается метод `rend()`. Перемещение по списку в обоих случаях выполняется одинаково — итератор увеличивается операцией инкремента. Однако результат совершенно разный: в первом цикле мы переходим к следующему элементу списка, а во втором — к предыдущему (рис. 5.1).

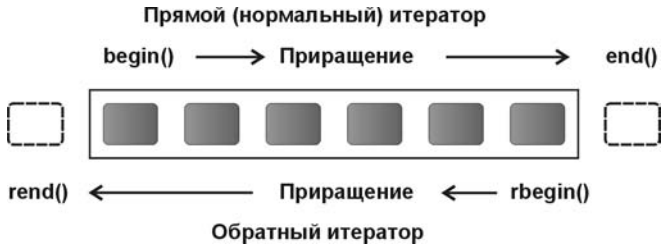

Рис. 5.1. Прямые и обратные итераторы

Пара итераторов всегда представляет собой *полуоткрытый интервал*:

```
[first, last)
```

Такая запись означает, что элемент `last` алгоритмом не обрабатывается — последним обрабатываемым элементом последовательности является элемент `last – 1`. Для пары прямых итераторов `[first, last)` итератор `first` обязательно должен быть левее итератора `last`, иначе последствия непредсказуемы. Для обратных — наоборот: начальный итератор должен быть правее конечного.

И наконец, обратите внимание на то, как выводится элемент списка на экран:

```
cout << *il << ' '; cout << *ril << ' ';
```

В *гл. 4* мы уже встречались с такой записью со звездочкой и знаем, что она означает операцию разыменования — получение значения. Таким образом, работая с итератором, мы имеем не один, а два объекта: сам итератор, и объект, который итератор представляет. В нашем случае — элемент контейнера-списка. Обращение к элементам контейнера не напрямую, а посредством итератора является косвенным обращением.

А теперь — некоторые технические подробности. Во-первых, любые виды итераторов бывают *константные* и *неконстантные*. Константные итераторы должны использоваться с контейнерами, описанными как `const` (например, в списке параметров). Во-вторых, в библиотеке STL определено 5 категорий итераторов: *входной* (`input`), *выходной* (`output`), *прямой* (`forward`), *двунаправленный* (`bidirectional`) и *произвольного доступа* (`random access`). Это не классы, и нам нет необходимости прописывать соответствующее слово при объявлении итератора.

Категории отличаются набором операций, которые разрешается применять к итератору. Для всех категорий итераторов разрешается использовать следующие операции (`i` и `j` — итераторы):

```
i++
++i
```

```
i = j
i == j
i != j
```

Остальные операции, определенные для разных категорий итераторов, приведены в табл. 5.2.

Таблица 5.2. *Категории итераторов*

Категория итератора	Операции	Контейнеры
Входной	`x = *i`	Все
Выходной	`*i = x`	Все
Прямой	`x = *i, *i = x`	Все
Двунаправленный	`x = *i, *i = x, --i, i--`	Все
Произвольный	`x = *i, *i = x, --i, i--, i + n, i – n, i += n, i -= n, i < j, i > j, i <= j, i > j`	Все, кроме `list`

Для всех итераторов, кроме произвольных, в библиотеке определены две функции: `distance`, которая выдает разность между двумя итераторами, и `advance`, которая реализует операцию `i += n`. Один из прототипов функции `distance` выглядит так:

```
template <class ForwardIterator, class Distance>
void distance(ForwardIterator first, ForwardIterator last, Distance &n);
```

В параметр `n` помещается значение типа `difference_type`, которое и является разностью итераторов `last – first`. Для вектора, например, эта разность представляет собой значение типа `unsigned int`.

Для двунаправленных и произвольных итераторов определены обратные (`reverse`) итераторы. Хотя они называются адаптерами, никакой разницы с точки зрения применения, по сравнению с обычными итераторами, нет. Для выходных итераторов также определены адаптеры — итераторы вставки:

- `back_insert_iterator` — используется совместно с функцией `back_inserter` (вставка в конец);
- `front_insert_iterator` — используется совместно с функцией `front_inserter` (вставка в начало);
- `insert_iterator` — используется совместно с функцией `inserter` (вставка перед заданным элементом).

Первые две функции имеют идентичные прототипы:

```
template <class Container>
back_insert_iterator<Container> back_inserter(Container& x)
```

и

```
template <class Container>
front_insert_iterator<Container> front_inserter(Container& x)
```

Имена говорят сами за себя и не нуждаются в пояснении. Третья функция несколько отличается:

```
template <class Container, class Iterator>
insert_iterator<Container> inserter(Container& x, Iterator i);
```

В данном случае в качестве параметра кроме контейнера передается еще итератор, задающий элемент, перед которым требуется выполнить вставку. Использование итераторов вставки поясним на простом примере. Обычно, чтобы скопировать один контейнер в другой, мы используем алгоритм `copy`. Причем мы должны задать одинаковую длину контейнеров, поскольку по умолчанию копирование работает в режиме замещения. Режим вставки работает при использовании итератора вставки:

```
vector<int> v(a, a+10);
list<int> L;
copy(v.begin(), v.end(), inserter(L, L.begin()));
```

А чтобы вставить в список `L` элементы вектора в обратном порядке, достаточно написать такой вызов:

```
copy(V.begin(), V.end(), front_inserter(L));
```

Есть еще и потоковые итераторы, которые позволяют работать с вводом/выводом, как с контейнерами. В многочисленных примерах из справочной системы Borland C++ Builder 6 приводится следующая последовательность операторов:

```
ostream_iterator<int, char> out(cout, " ");
copy(v1.begin(),v1.end(),out);
```

Сначала здесь объявляется итератор вывода `out`, а затем он используется в алгоритме копирования для вывода значений элементов вектора на экран. В угловых скобках в объявлении итератора прописан тип выводимых значений и тип элемента, разделяющего эти значения. В круглых скобках заданы оператор `cout` и пробел, разделяющий элемент.

Можно непосредственно задать итератор вывода в качестве параметра, например:

```
fill_n(ostream_iterator<int,char>(cout," "), 3, 5);
```

Этот оператор выведет на экран три пятерки. А оператор

```
fill_n(ostream_iterator<char,char>(cout, '-'), 30, '-');
```

"нарисует" на экране 60 минусов.

В справочной системе Borland C++ Builder 6 есть пример (его текст приведен в листинге 5.14), поясняющий работу итератора вывода. Сначала с помощью итератора на экран выводится поясняющая строка, а затем вычисляется и выводится сумма элементов дека.

Листинг 5.14. Использование итератора вывода

```
#include <iterator>    //  обязательно указывать
#include <numeric>
#include <deque>
#include <iostream>
using namespace std;
int main()
{  int arr[4] = { 3, 4, 7, 8 };
   int total = 0;
   deque <int> d(arr + 0, arr + 4);
    //  итератор вывода выводит на экран поясняющую строку
   copy (d.begin(), d.end() — 1,
   ostream_iterator <int, char> (cout, " + "));
   cout << *(d.end() — 1) << " = " <<
   accumulate(d.begin(), d.end(), total) << endl;
   return 0;
}
```

Итераторы могут быть действительными (когда итератору соответствует элемент контейнера) и недействительными (когда итератору не соответствует никакой элемент контейнера). Итератор может стать недействительным по трем причинам:

- итератор не был инициализирован;
- итератор равен `end()`;
- контейнер, с которым связан итератор, изменил размеры или вовсе уничтожен.

Для работы с итераторами необходимо прописывать в программе оператор:

```
#include <iterator>
```

Поиск в контейнере

По традиции алгоритмы делят на несколько классов:

- алгоритмы, не модифицирующие последовательность — это алгоритмы поиска и подсчета элементов;

- алгоритмы, модифицирующие последовательность — это алгоритмы, изменяющие либо значения элементов, либо состав элементов контейнера;
- алгоритмы сортировки;
- разные — алгоритмы, которые не входят в первые три класса, например, алгоритмы для работы с "кучей" или численные алгоритмы.

Многие контейнеры обладают методами, функционально эквивалентными стандартным алгоритмам. Например, контейнер-список имеет метод `sort`, выполняющий ту же работу, что и универсальный алгоритм `sort`. Такая избыточность имеет две причины: во-первых, стандартный алгоритм в подавляющем большинстве случаев более универсален, в частности может работать и с массивом, и с контейнером; во-вторых, метод контейнера часто реализован более эффективно для данного типа контейнера.

Рассмотрим варианты алгоритмов поиска и сортировки, которых в стандартной библиотеке шаблонов очень много (табл. 5.3).

Таблица 5.3. *Алгоритмы сортировки и поиска*

Алгоритм	Тип	Назначение алгоритма
`find`	Несорт	Находит первое слева заданное значение
`find_first_of`	Несорт	Находит первое вхождение значения из одной последовательности в другой
`find_end`	Несорт	Находит последнее вхождение значения из одной последовательности в другой
`find_if`	Несорт	Находит первое значение, удовлетворяющее заданному предикату
`search`	Несорт	Ищет первое вхождение одной последовательности в другой
`search_n`	Несорт	Ищет `n`-е вхождение одной последовательности в другой
`count`	Несорт	Подсчитывает количество элемента в последовательности
`count_if`	Несорт	Подсчитывает количество элемента в последовательности, удовлетворяющего заданному предикату
`equal`	Несорт	Попарное сравнение двух последовательностей
`lower_bound`	Сорт	Находит в сортированной последовательности первый элемент не меньше заданного
`upper_bound`	Сорт	Находит в сортированной последовательности первый элемент больше заданного

Таблица 5.3 *(окончание)*

Алгоритм	Тип	Назначение алгоритма
binary_search	Сорт	Двоичный поиск заданного элемента в последовательности
equal_range	Сорт	Находит в заданной последовательности самую длинную последовательность, содержащую заданный элемент
sort	Несорт	Сортировка последовательности
partial_sort	Несорт	Сортирует часть последовательности
merge	Сорт	Сливает две сортированные последовательности

Начнем с поиска минимального и максимального элемента в контейнере. Эти алгоритмы работают с неупорядоченным контейнером. Выполним небольшую программу, текст которой приведен в листинге 5.15.

Листинг 5.15. Поиск максимального и минимального элемента контейнера

```
#include <iostream>
#include <algorithm>
#include <list>
#include <cstdlib>
#include <ctime>
using namespace std;
#include <Rus.h>
int main()
{  double elem; int i; const int nn = 20; double mu[nn];
    //  заполнение массива случайными числами
   time_t t; srand((unsigned) time(&t));
   for (i = 0; i < nn; ++i) mu[i] = (rand() % 1000);
   list <double> u(mu, mu + nn);                          //  список
   list <double> :: iterator iu;                          //  итератор
    //  вывод списка на экран
   for (iu = u.begin(); iu != u.end(); ++iu) cout << *iu << ' ';
   cout << endl;
   elem = *min_element(u.begin(), u.end());               //  1
   cout << elem << endl;
   cout << *max_element(u.begin(), u.end()) << endl;      //  2
   cout << *min_element(mu, mu + nn) << endl;             //  3
   return 0;
}
```

В этой программе сначала объявляется и заполняется случайными числами массив mu, затем объявляется, инициализируется и выводится на экран список u. Строки 1 и 2 показывают применение стандартных алгоритмов поиска максимального и минимального элемента в списке. Если мы хотим обрабатывать массив, то вместо итераторов нужно задавать параметры-массивы в указательной форме, что мы и наблюдаем в строке 3. Звездочка впереди должна проставляться, т. к. эти алгоритмы возвращают итератор. Точно так же выглядит обращение и для вектора, и для дека.

Алгоритмы поиска минимального и максимального элемента обладают интересным свойством, с которым мы еще не сталкивались: вызов может стоять слева от знака присваивания. Добавим в листинг 5.15 следующие строки:

```
*min_element(u.begin(), u.end()) = 400;
*max_element(mu, mu + nn) = 200;
```

После этого на месте минимального элемента списка будет стоять 400, а на месте максимального элемента массива — 200. Таким свойством обладают все алгоритмы, возвращающие итератор. Обратите внимание на то, что эти алгоритмы написаны средствами стандартного C++ — значит мы тоже можем писать такие функции. Как это делается, мы рассмотрим в *гл. 7*.

Алгоритмы сортировки и поиска реализованы так, что они могут работать с любой последовательностью элементов контейнера. Как всегда, аргументами является пара итераторов, поэтому эти алгоритмы могут обработать и весь контейнер целиком. Чтобы немного разобраться с этими алгоритмами, а также с некоторыми другими, как обычно, рассмотрим простой пример (его текст приведен в листинге 5.16). Заголовки мы не указываем, чтобы не занимать место.

Листинг 5.16. Поиск в списке

```
bool isPrime(int number)     //  предикат — простое число
{  number = abs(number);     //  отрицательные — не принимаем
   // 0 и 1 — не являются простыми числами
   if (number == 0 || number == 1) return false;
   int divisor;               //  ищем делитель, который делит без остатка
   for (divisor = number / 2; number%divisor != 0; --divisor);
   return divisor == 1;       //  если такой — только 1, то число простое
}
int main()
{  list<int> L;                  //  список
   list<int> :: iterator pL;     //  итератор
   // заполняем список целыми числами от 20 до 40
   for (int i = 20; i <= 40; ++i) L.push_back(i);
```

```
    for (pL = L.begin(); pL != L.end(); ++pL) cout << setw(3) << *pL;
    cout << endl;
    pL = find(L.begin(), L.end(), 31); /* ищем число 31  */
    /* реверсируем последовательность от 31 до конца списка */
    reverse(pL, L.end());
    for (pL = L.begin(); pL != L.end(); ++pL) cout << setw(3) << *pL;
    cout << endl;
    list<int> :: iterator pos;
    pos = find_if(L.begin(), L.end(), isPrime);    //  ищем простое число
    if (pos != L.end()) cout << *pos << " — первое простое число" << endl;
    else cout << " — нет простого числа" << endl;  //  не нашли
    //  считаем простые числа в списке
    cout << count_if(L.begin(), L.end(), isPrime) << endl;
    L.sort();    //  сортировка списка
    for (pL = L.begin(); pL != L.end(); ++pL) cout << setw(3) << *pL;
    cout << endl;
    return 0;
}
```

В этой программе сначала, как обычно, объявляется список и заполняется числовым рядом от 20 до 40. Следует обратить внимание на оператор вывода:

```
cout << setw(3) << *pL;
```

Здесь мы впервые использовали манипулятор, задающий ширину вывода значения — каждый элемент списка `L` займет на экране 3 позиции. Чтобы такая запись работала, необходимо задать в программе заголовок:

```
#include <iomanip>
```

Манипуляторов в стандартной библиотеке определено достаточно много, они позволяют сформатировать вывод.

Далее в списке разыскивается значение `31` и выполняется реверс "хвоста" списка, начиная с элемента `31`. Если исходный список был таким:

```
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

то после реверса список станет таким:

```
20 21 22 23 24 25 26 27 28 29 30 40 39 38 37 36 35 34 33 32 31
```

Мы, в отличие от примера в листинге 5.13, физически изменили порядок следования элементов на обратный. Список обладает и собственным методом `reverse`, однако алгоритм более универсален — метод может обработать только весь список целиком.

У нас определена функция-предикат `isPrime`, распознающая простое число. Оператор

```
pos = find_if(L.begin(), L.end(), isPrime);
```

выполняет поиск в списке первого простого числа. В нашем случае это число `23`. Если бы простого числа в списке не оказалось, то итератор `pos` получил значение `L.end()` — вне списка. Предикат передается стандартному алгоритму `find_if` в качестве параметра. Этот пример хорошо иллюстрирует идею универсальности алгоритмов — функция `find_if` способна обрабатывать любую последовательность элементов, как массива, так и любого последовательного контейнера, тип элементов может быть любым, критерий поиска (предикат) — тоже произвольный. Более того, т. к. функция возвращает итератор, мы можем использовать ее слева от операции присваивания:

```
*find_if(L.begin(), L.end(), isPrime) = 0;
```

Этот оператор заменит первое простое число нулем.

Тот же предикат используется для подсчета количества простых значений в списке с помощью алгоритма `count_if`.

```
cout << count_if(L.begin(), L.end(), isPrime) << endl;
```

Как видите, по сравнению с алгоритмом поиска изменилось только имя функции. Единообразие прототипов — одна из самых привлекательных черт библиотеки STL.

Во многих случаях мы без всякой переделки могли бы заменить список на вектор, дек или массив. А вот универсальный алгоритм сортировки со списком не работает, поэтому нам пришлось вызвать собственный метод списка. Собственный метод менее универсален — обрабатывает весь список целиком, тогда как алгоритм `sort` может обрабатывать любую последовательность, заданную парой итераторов.

Сортировки

Применение алгоритмов сортировки не представляет труда. Посмотрите, как можно отсортировать и объединить два массива (текст примера приведен в листинге 5.17).

Листинг 5.17. Сортировки и объединение массивов

```
double mu[10], nu[10];
time_t t; srand((unsigned) time(&t));
for (i = 0; i < 10; ++i) mu[i] = (rand() % 100);
for (i = 0; i < 10; ++i) cout << mu[i]<< ' '; cout << endl;
for (i = 0; i < 10; ++i) nu[i] = (rand() % 100);
for (i = 0; i < 10; ++i) cout << setw(3) << nu[i]; cout << endl;
```

```
sort(mu, mu + 10);                          //  2
sort(nu, nu + 10);                          //  3
double ru[20];
merge(mu, mu + 10, nu, nu + 10, ru);        //  4
for (i = 0; i < 20; ++i) cout << setw(3) << ru[i]; cout << endl;
```

Сначала мы, как всегда, заполняем массивы случайными числами. Затем сортируем оба массива (строки 2 и 3) и объединяем сортированные массивы в новый массив (строка 4). В результате выполнения этого фрагмента программы на экране появится следующее:

```
56 5 83 12 33 39 68 37 91 83                                  //  массив mu
27 0 0 83 97 0 16 0 37 67                                     //  массив nu
0 0 0 0 5 12 16 27 33 37 37 39 56 67 68 83 83 83 91 97        //  массив ru
```

С помощью алгоритма sort мы, естественно, можем отсортировать не только весь массив (контейнер), но и его часть — надо лишь задать соответствующие параметры. Однако в стандартной библиотеке реализовано несколько алгоритмов частичной сортировки. Они нужны в том случае, если контейнер большой, а нас интересуют не все элементы, а "горячая десятка", например, самых маленьких.

Обработка контейнеров и функциональные объекты

В результате выполнения программы из листинга 5.17 в массиве ru получилось объединение двух массивов — mu и nu. В объединенном массиве есть повторяющиеся элементы. Чтобы оставить в массиве единственную копию повторяющегося элемента, можно применить алгоритм unique:

```
unique(ru, ru + 20);
for (i = 0; i < 20; ++i) cout << setw(3) << ru[i]; cout << endl;
```

Как видно по результату выполнения, этот алгоритм просто перемещает элементы массива вперед к началу:

```
0 0 0 0 0 6 23 34 36 43 58 59 73 73 78 80 83 89 90 91
0 6 23 34 36 43 58 59 73 78 80 83 89 90 91 80 83 89 90 91
```

Выполним ту же работу с вектором. Мы создадим вектор из объединенного массива ru до удаления повторяющихся элементов, а потом применим алгоритм unique. Тем самым выясним длину вектора после выполнения алгоритма:

```
vector<double> V(ru, ru + 20);
unique(V.begin(), V.end());
for (i = 0; i < V.size(); ++i) cout << setw(3) << V[i]; cout << endl;
```

К сожалению, и с векторами этот алгоритм поступает точно так же, как с массивами — просто перемещает элементы в начало, не удаляя их.

Аналогично поступает и алгоритм `remove` — в библиотеке четыре разновидности этого алгоритма: "простой" `remove`, с копированием `remove_copy`, условный `remove_if` и условный с копированием `remove_copy_if`. Тем не менее эти алгоритмы, так же как и алгоритмы `unique`, возвращают итератор на логический конец данных. Поэтому мы можем воспользоваться методом контейнера `erase`, чтобы физически удалить элементы из контейнера. В качестве примера воспользуемся тем же массивом `ru` и вектором `V`:

```
vector<double> V(ru, ru + 20);
remove(ru, ru + 20, 0);
for (i = 0; i < 20; ++i) cout << setw(3) << ru[i]; cout << endl;
vector<double> :: iterator pos = remove_if(V.begin(), V.end(), isPrime);
for (i = 0; i < V.size(); ++i) cout << setw(3) << V[i]; cout << endl;
V.erase(pos, V.end());
for (i = 0; i < V.size(); ++i) cout << setw(3) << V[i]; cout << endl;
```

Если исходный массив `ru` содержал следующие элементы:

```
0 0 0 0 0 0 7 14 15 26 33 37 38 47 61 62 65 68 69 85
```

то после выполнения `remove` массив приобретет такой вид:

```
7 14 15 26 33 37 38 47 61 62 65 68 69 85 61 62 65 68 69 85
```

Как видим, элементы сдвинулись вперед, затирая начало массива, однако "хвост" массива остался без изменения. Точно так же алгоритмы группы `remove` обрабатывают контейнеры. Вектор `V` после выполнения `remove_if` будет содержать следующие элементы:

```
0 0 0 0 0 0 14 15 26 33 38 62 65 68 69 85 65 68 69 85
```

Длина опять осталась неизменной — элементы сдвинулись вперед на место элементов — простых чисел. И только после исполнения метода `erase` элементы физически удаляются из контейнера, что мы и наблюдаем на экране:

```
0 0 0 0 0 0 14 15 26 33 38 62 65 68 69 85
```

Алгоритмы, подобные `unique`, `remove`, `copy`, `reverse`, которые мы уже использовали, модифицируют последовательности. В стандартной библиотеке STL реализовано достаточно много таких алгоритмов. Некоторые из них представлены в табл. 5.4.

Таблица 5.4. *Алгоритмы, изменяющие последовательность*

Алгоритм	Назначение алгоритма
`copy`	Копирует последовательность, начиная с первого элемента
`copy_backward`	Копирует последовательность, начиная с последнего элемента

Таблица 5.4 *(окончание)*

Алгоритм	Назначение алгоритма
`fill`	Заполняет последовательность заданным значением
`fill_n`	Замена первых n элементов заданным значением
`generate`	Заменяет элементы контейнера результатом заданной операции
`generate_n`	Замена первых n элементов результатом операции
`remove`	"Удаление" заданных элементов последовательности
`remove_if`	"Удаление" элементов, удовлетворяющих предикату
`replace`	Замена элементов заданным значением
`replace_if`	Замена элементов, удовлетворяющих условию
`transform`	Выполнение операции над каждым элементом последовательности
`unique`	"Удаление" повторяющихся элементов
`unique_copy`	"Удаление" повторяющихся элементов с копированием в новую последовательность
`reverse`	Обращение последовательности
`reverse_copy`	Обращение последовательности с копированием

Эти, и большинство других алгоритмов в качестве одного из аргументов принимают функцию, которая вызывается алгоритмом для обработки входной (или выходной) последовательности. Например, мы часто использовали предикат для отбора требуемых элементов контейнера. Однако в большинстве случаев нам нет необходимости писать свою функцию: в библиотеке STL определен ряд *функциональных объектов*[1], которые разрешается использовать в качестве параметра вместо функции. Функциональные объекты — это четвертый краеугольный "камень" стандартной библиотеки. Стандартные функциональные объекты перечислены в табл. 5.5 — реализованы все арифметические операции, операции сравнения и логические операции.

Таблица 5.5. *Стандартные функциональные объекты*

Функциональный объект	Реализуемая операция	Значение
`plus<T>`	`x + y`	Сложение
`minus<T>`	`x – y`	Вычитание

[1] Вернее, в библиотеке определены шаблоны функциональных объектов.

Таблица 5.5 *(окончание)*

Функциональный объект	Реализуемая операция	Значение
`multiplies<T>`	`x * y`	Умножение
`divides<T>`	`x / y`	Деление
`modulus<T>`	`x % y`	Остаток от деления
`negate<T>`	`- x`	Изменение знака
`equal_to<T>`	`x == y`	Равенство
`not_equal_to<T>`	`x != y`	Неравенство
`greater<T>`	`x > y`	Больше
`less<T>`	`x < y`	Меньше
`greater_equal<T>`	`x >= y`	Больше или равно
`less_equal <T>`	`x <= y`	Меньше или равно
`logical_and`	`x && y`	Логическое И
`logical_or`	`x || y`	Логическое ИЛИ
`logical_not`	`!x`	Логическое отрицание

Давайте отсортируем вектор целых чисел по убыванию. До сих пор мы использовали алгоритм сортировки только в одном варианте: сортировка по умолчанию выполняется в порядке возрастания. Для сортировки по убыванию можно написать функцию сравнения:

```
bool compare(int x, int y) { return x > y; }
```

Тогда сортировка по убыванию вектора `V` выполняется так:

```
sort(V.begin(), V.end(), compare);
```

Однако нам нет необходимости писать подобную функцию сравнения, т. к. в библиотеке реализован нужный нам функциональный объект `greater<T>`. Поэтому выполнение сортировки по убыванию можно делать так:

```
sort(V.begin(), V.end(), greater<int>());
```

Чтобы использовать стандартные функциональные объекты, надо в программе прописать оператор:

```
#include <functional>
```

Давайте рассмотрим еще несколько простых примеров. Мы уже использовали стандартный алгоритм `accumulate` для вычисления суммы элементов вектора (см. листинг 5.11). Задействовав арифметические функциональные объ-

екты, можно вычислять не только сумму, но и, например, произведение элементов вектора:

```
vector<double> v;    //  как-то заполняем вектор
double p = accumulate(v.begin(), v.end(), 1.0, multiplies<double>());
```

До сих пор мы заполняли массивы случайными числами, прописывая в программе соответствующий цикл. Очевидно, подобные действия приходится делать довольно часто, поэтому разработчики STL позаботились об этом, реализовав алгоритмы группы `generate`. Определив некоторую функцию, например:

```
double f(){ return (rand() % 100); }
```

мы можем заполнить контейнер с помощью одного из этих алгоритмов:

```
double mu[10], nu[10]; time_t t; srand((unsigned) time(&t));
generate(mu, mu + 10, f);
```

Аналогично можно использовать и алгоритм `transform`, который на самом деле не изменяет исходной последовательности, а записывает результат преобразования в другую последовательность. Кроме того, алгоритм имеет две формы: одна работает с единственной последовательностью, применяя к каждому элементу унарную операцию, другая — обрабатывает две, элементы которых являются аргументами заданной бинарной операции. Воспользуемся только что заполненным массивом `mu`:

```
transform(mu, mu + 10, mu, negate<double>());
fill(nu, nu + 10, 1);    //  заполнили второй массив
transform(mu, mu + 10, nu, nu, plus<double>());
```

Здесь мы сначала делаем элементы массива `mu` отрицательными (унарный `transform`), затем заполняем второй массив единицами, а потом складываем поэлементно два массива (бинарный `transform`), записывая результат на место второго. Алгоритм `transform` позволяет использовать и функцию. Например, мы хотим умножить все элементы массива `nu` на `2`:

```
double f2(double a) { return a*2; }
transform(nu, nu + 10, nu, f2);
```

Как видите, для этого достаточно написать простую функцию с одним аргументом (алгоритм-то унарный!) и передать ее как аргумент. Аналогично можно обрабатывать два массива, если написать функцию с двумя параметрами:

```
double ff(double a, double b) { return a*2-b;}
fill(nu, nu + 10, 1);    //  заполнили второй массив
transform(mu, mu + 10, nu, nu, ff);
```

Если в массиве `mu` были сгенерированы значения:

```
30 91 94 82 66 54 56 56 31 95
```

то в результате получим:

```
59 181 187 163 131 107 111 111 61 189
```

Мораль такова: пользуйтесь контейнерами и алгоритмами стандартной библиотеки везде, где только возможно. Это здорово облегчает написание программы и повышает надежность — алгоритмы-то отлажены!

ГЛАВА 6

Ввод и вывод в C++

Процесс получения программой информации из внешнего мира называется *вводом*, а отправка информации из программы во внешний мир — это *вывод*. Естественно, ввод осуществляется с некоторого *устройства ввода*, а вывод выполняется на *устройство вывода*. Любая полезная программа всегда выполняет ввод и вывод информации. Во всех наших программах ввод выполнялся с клавиатуры, а вывод — на экран. Однако в состав компьютера входят и другие устройства ввода/вывода, например магнитные диски и принтер. Выполнение ввода/вывода естественно зависит от того, с каким устройством приходится работать. Кроме того, ввод/вывод зависит и от операционной системы, которая осуществляет управление устройствами с помощью драйверов. *Файловая система*, которая является неотъемлемой частью любой операционной системы, осуществляет управление информацией.

По этим причинам средства ввода/вывода в языках программирования всегда были машинно- и системно-зависимыми. Авторы языка C, как и во многих других случаях, приняли новаторские решения, которые во многом способствовали успеху и языка, и операционной системы UNIX. Во-первых, средства ввода/вывода были отделены от языка и вынесены в отдельную библиотеку. Эта библиотека получила имя stdio (стандартная библиотека ввода/вывода) и входит в стандарты C и C++. Во-вторых, удалось разработать и реализовать концепцию независимого от устройств ввода/вывода. Программа на C не имеет дело ни с устройствами, ни с файлами — она работает с *потоками*. Ввод информации осуществляется из *входного потока*, вывод программа производит в *выходной поток*. Естественно, есть возможность связать поток с устройством или с файлом.

В C++ эти концепции получили дальнейшее развитие в виде объектно-ориентированной библиотеки ввода/вывода iostream. К сожалению, ввод/вывод — это самая модифицируемая часть C++. Если библиотека stdio

уже давно стала стандартом, то реализация библиотеки iostream изменялась вплоть до принятия стандарта. Да и после принятия должно было пройти некоторое время, чтобы в очередных версиях систем программирования появилась реализация, полностью соответствующая стандарту. Некоторые вопросы стандарт оставляет "на усмотрение" разработчиков. Не всегда в широко доступной литературе описана именно версия стандарта. Например, в [29] описана реализация ввода/вывода Visual C++ 6, которая не совсем соответствует стандарту. Поэтому даже автор языка Б. Страуструп советует [37, с. 705]: "Насчет деталей сверьтесь, пожалуйста, с руководством по вашей системе — и поэкспериментируйте". Изложение в данной главе основано на [37] и все примеры проверены в системах Borland C++ Builder 6 и Visual C++ 6 с установленной библиотекой STLport 4.5.3.

Стандартные потоки в C++

Ввод/вывод в C++ основан на тех же принципах, что и ввод/вывод в C. Стандартная библиотека предоставляет программисту ряд классов, которые реализуют три вида потоков: стандартные, файловые и строковые потоки. Программа извлекает данные из входного потока и помещает в выходной. Файловые потоки, как правило, бывают *двунаправленными*. Строковые потоки — это реализация поточного механизма для оператора `string`. Эти потоки также бывают двунаправленными. Стандартные потоки нам уже знакомы. Заголовок `<iostream>` содержит описания классов ввода/вывода и четыре системных объекта:

- `cin` — стандартный поток ввода; по умолчанию связан с клавиатурой;
- `cout` — стандартный поток вывода; по умолчанию связан с экраном;
- `clog` — стандартный поток вывода; по умолчанию связан с экраном;
- `cerr` — стандартный поток вывода; по умолчанию связан с экраном.

Объект `cout` предназначен для "нормального" вывода, а объекты `cerr` и `clog` — для вывода сообщений об ошибках. Использовать их можно точно так же, как и `cout`:

```
cerr << "Это сообщение об ошибке!" << endl;
clog << "И это тоже сообщение об ошибке!" << endl;
```

"Ошибочные" объекты позволяют различать в тексте программы "нормальный" вывод и сообщения об ошибках. Вообще-то говоря, мы можем вместо `cout` использовать либо `cerr`, либо `clog`, однако лучше все-таки придерживаться общепринятых соглашений и "нормальный" вывод направлять в `cout`, а сообщения об ошибках — в `cerr` или `clog`.

Мы уже неоднократно пользовались стандартным потоком вывода в простых вариантах, поэтому не будем повторяться. Однако Б. Страуструп [37,

с. 671] замечает: "то, что разработчик прикладных программ считает выводом, на самом деле является преобразованием объектов некоторого типа ... в последовательность символов". Таким образом, становится ясно, что поток — это поток символов. Операция ввода `>>` извлекает некоторую последовательность символов из входного потока `cin` и преобразует в объект некоторого типа (например, в `int` или `double` — это зависит от переменной, куда должен быть помещен результат). Операция вывода `<<` переводит объект в последовательность символов и помещает символы в выходной поток `cout`. При работе со стандартными потоками преобразование выполняется для всех числовых типов (вспомним функцию перевода строки символов в целое число — см. листинг 3.12).

Ввод данных

Рассмотрим подробнее ввод числовых данных. Допустим, нам необходимо ввести целое число `31` и дробное число `12.78`. Мы пишем в программе такие операторы:

```
int a;
double b;
cin >> a >> b;
```

Тогда числа можно задавать так:

```
31<enter>
12.78<enter>
```

или так:

```
31<пробел>12.78<enter>
```

Как видим, ввод должен заканчиваться клавишей <Enter>. Мы могли бы написать операторы ввода и следующим образом:

```
cin >> a;
cin >> b;
```

Это никак не влияет на то, какую последовательность символов должен вводить пользователь, и сколько раз будет нажата клавиша <Enter> — все, что описано выше, работает и в данном варианте.

Если в последовательности символов встречаются недопустимые для данного типа символы, то ввод не выполняется, например:

```
31r12.78<enter>
```

Число `31` попадет в целую переменную, а дробное число — нет. Если недопустимый символ будет перед первым числом, то ввод вообще не выполнится ни для первого числа, ни для второго. Таким образом, разделителем чисел в последовательности символов является пробел. Стандартными сим-

волами-разделителями являются также табуляция, новая строка, новая страница и возврат каретки. Символы-разделители пропускаются операцией >> при вводе, поэтому, как пишет Б. Страуструп [37, с. 681], мы могли бы заполнять вектор таким образом (текст примера приведен в листинге 6.1).

Листинг 6.1. Ввод дробных чисел в вектор

```
int read_double(vector <double> &v)
{ int i = 0; while (i < v.size() && cin >> v[i]) i++; return i; }
```

Вместо входного потока можно использовать итератор ввода, связав его с входным потоком (текст примера приведен в листинге 6.2).

Листинг 6.2. Ввод дробных чисел в вектор итератором ввода

```
int read_double(vector <double> &v)
{  istream_iterator<double>is(cin);    //  объявление и связывание
   istream_iterator<double>eof;        //  завершающий итератор
   copy(is, eof, back_inserter(v));
   return v.size();
}
```

Алгоритм `copy` использует пару однотипных итераторов ввода: первый итератор связывается с входным потоком, второй необходим для обозначения окончания ввода. Имя `eof` является "говорящим" и расшифровывается как "end of file" — конец файла. На самом деле имя может быть любым, например, `dd`. Процесс ввода будет продолжаться до тех пор, пока пользователь не нажмет комбинацию клавиш <Ctrl>+<Z>, а потом клавишу <Enter>. Например, чтобы ввести 5 чисел, пользователь может набрать на клавиатуре следующую последовательность символов:

```
1<пробел>2<пробел>3<пробел>4<пробел>5<пробел><Ctrl>+<Z><Enter>
```

Как обычно, вместо любого пробела можно нажимать клавишу <Enter>.

Если у нас во входном потоке встречаются символы, отличающиеся от стандартных символов-разделителей, мы должны реализовать в программе ввод символа. Пусть у нас *макет ввода* представляет собой следующее:

```
<целое><символ><дробное>
```

Надо объявить соответствующие переменные в программе:

```
int a;
char ch;
double b;
```

Тогда строку

```
31r12.78
```

мы можем вводить оператором:

```
cin >> a >> ch >> b;
```

На месте символа может быть задан любой символ (в т. ч. и пробел), который попадет в переменную `ch`. При вводе отдельного символа мы тоже должны нажимать клавишу <Enter>. Если нам необходимо ввести с клавиатуры несколько символов в цикле:

```
char ch;
for (int i = 0; i < 10; ++i) cin >> ch;
```

то мы можем нажать клавишу <Enter> либо один раз, либо два раза, либо после каждого символа. Мы можем даже нажать гораздо больше 10-ти клавиш-символов, однако после нажатия <Enter> в программу попадут только первые 10 из них. Чтобы в этом убедиться, выполним тот же цикл с оператором вывода:

```
for (int i = 0; i < 10; ++i) { cin >> ch; cout << ch; }
```

Аналогичная картина получается и при вводе чисел: мы можем вводить достаточно длинную последовательность символов-цифр, разделяемых пробелами, но в программу попадет (после нажатия клавиши <Enter>) ровно столько чисел, сколько переменных написано в операторах ввода.

Такое несколько странное с точки зрения здравого смысла поведение (зачем разрешать ввод лишних символов, если они не используются) объясняется тем, что потоки буферизованы. Это означает, что на самом деле символы, нажимаемые пользователем на клавиатуре, сначала попадают в промежуточную память, выделяемую системой, которая называется *буфером ввода*, и только оттуда (после преобразования) выполняется перенос в переменные нашей программы. Аналогично работает и вывод — после преобразования символы заносятся в буфер, а уже из буфера выводятся на устройство. Схема работы ввода/вывода с буферизацией изображена на рис. 6.1.

Буферизация ввода/вывода преследует две цели:

- сделать прикладные программы более независимыми от устройств ввода/вывода;
- повысить быстродействие прикладных программ.

Из стандартных потоков не буферизованным является только `cerr`. Буферизация потоков управляема, но в простых задачах редко бывает нужна.

Ввод строк

В *гл. 3* мы выяснили, что ввод символьных массивов операцией `>>` выполняется только до пробела. Аналогичная картина наблюдается и при вводе

Рис. 6.1. Буферизация ввода/вывода

строк типа `string`. При вводе массивов символов и строк операцией `>>` первые пробелы пропускаются, и ввод значащих символов выполняется до первого пробела. Но стандартная библиотека предоставляет средства ввода строк вместе с пробелами.

Для ввода последовательности символов и размещения их в символьном массиве библиотека ввода/вывода включает три функции-метода: `read`, `get` и `getline`. Функция `read` не очень удобна для ввода символов. Она используется, в основном, для того, чтобы на ее основе программировать ввод более высокого уровня. Значительно удобнее функция-метод `get`. Она позволяет в массив из `n` элементов вводить не более `n – 1` символов, следит за символом завершения ввода и проставляет после введенных символов нулевой байт. Пусть в нашей программе объявлен массив символов

```
char s[100] = {0};
```

проинициализированный нулями, чтобы не было проблем с выводом. Для ввода любой строки в массив `s` нам достаточно написать в программе вызов:

```
cin.get(s, 100);
```

Завершается ввод, как всегда, клавишей <Enter>, при нажатии которой во входном потоке формируется символ `'\n'`, а в массив символов на соответ-

ствующее место попадает нулевой байт. Чтобы можно было корректно продолжить ввод (следующей строки), символ '\n' надо удалить из входного потока. Это делается методом ignore. Таким образом, ввод одной строки в массив s выполняется так:

```
cin.get(s, 100);
cin.ignore();
```

Символ '\n' является символом завершения ввода строки по умолчанию, однако мы можем задать любой символ завершения, например ';'.

```
cin.get(s, 100, ';');
cin.ignore();
```

Правда, для окончания ввода все равно придется нажимать клавишу <Enter>. Как видим, функция get также не слишком удобна. Функция-метод getline работает при вводе символов аналогично функции get, но еще и удаляет из входного потока тот самый символ-завершитель строки, так что ignore вызывать нет необходимости. Наши примеры с использованием функции getline выглядят так:

```
cin.getline(s, 100);
cin.getline(s, 100, ';');
```

Важно отметить, что введенная с клавиатуры строка не нуждается в перекодировке. Перекодировать необходимо только константы, которые мы набирали в редакторе интегрированной среды.

Для ввода строк в классе string имеется собственный метод getline, вызывать который надо так:

```
string s;
getline(cin, s);    //  ввод из стандартного потока
```

Очевидно, это сделано для большей общности, т. к. на месте первого параметра может стоять не только стандартный поток cin, а, например, поток, связанный с файлом. Так же, как и массивы символов, вводимые строки нет необходимости перекодировать при выводе.

Потоки и файлы

Даже если вы никогда не написали ни одной программы, как пользователю вам должно быть известно, что вся информация сохраняется на диске в виде файлов. *Файл* — это поименованная "порция" информации. Файлы необходимы для долговременного хранения данных (и программ) и являются самой простой формой связи между программами: файл, записанный одной программой, может прочитать другая программа.

Для управления множеством файлов в состав операционной системы входит *файловая система*. Именно файловая система определяет, каким образом именуются файлы и где они размещаются. В разных операционных системах — разные правила именования файлов. Даже в разных версиях одной операционной системы эти правила бывают различны! Более того, в рамках одной операционной системы может использоваться несколько файловых систем, в которых правила именования файлов различаются.

Для работы с файловыми потоками мы должны знать, как именуются файлы в той операционной системе, под управлением которой будет работать программа. Использование файлов в программе возможно только при выполнении следующих операций:

- создание потока;
- открытие файла и связывание его с потоком;
- обмен (ввод/вывод);
- закрытие файла;
- уничтожение потока.

Файловый поток является обычной переменной в программе, которая имеет область видимости и "время жизни" в соответствии с объявлением. Эта переменная не имеет никакого отношения к файлам на диске — необходимо как-то указать, что объявленная переменная связана с некоторым файлом. Эта связь осуществляется по имени файла либо при создании потока, либо при открытии файла.

Примечание

В "доисторические" времена операции открытия и закрытия выполнялись для файлов — потоков еще не придумали. Поскольку в программе мы работаем с потоковой переменной, то сейчас можно говорить об открытии и закрытии как файлов, так и потоков. Мы будем употреблять и то, и другое выражение.

Обмен выполняется функциями-методами класса файлового потока. Закрытие файла выполняется либо методом класса, либо при уничтожении потока.

Каталоги

Поскольку мы работаем в Windows, нам надо разобраться, как именуются файлы в этой операционной системе и как представляются имена файлов в программе на C++. Кроме того, очень важно научиться управлять размещением своих файлов в нужных каталогах.

В программе на C++ имя файла представляется константой-строкой. Или можно объявить переменную — символьный массив, в который помещается

строка-имя файла. Тип `string` использовать нельзя. Имя файла должно удовлетворять следующим ограничениям:

- нельзя использовать символы >, <, :, ", |;
- имя может содержать пробелы и русские буквы, однако при выводе на экран требуется перекодировка;
- имя нечувствительно к регистру, однако регистр в имени сохраняется. Если файл был создан с именем file, то в таком виде оно и будет отображаться, но к файлу можно обращаться и по имена FILE, и по имени File.

Напомним, что собственно имя файла состоит из имени и расширения, которое может и отсутствовать. Если оно есть, то отделяется от имени символом "точка". Расширение часто (но не всегда) определяет тип файла. Полное имя файла включает еще *путь к файлу*. Путь — это диск и каталог, в котором размещен файл. Длина полного имени ограничена константой Windows `MAX_PATH`.

Примечание

При работе в системе Borland C++ 3.1 длина имени файла (без каталогов) не должна превосходить 8 символов. Так было реализовано в MSDOS и Windows 3.1. Более поздние версии Windows (95, 98, 2000, XP и др.) наряду с длинными именами предоставляют и сокращенные до 8 символов имена.

Диски обозначаются английской буквой **A**–**Z** (или **a**–**z**) с двоеточием. Буква **A** обычно обозначает дискету. Вложенные каталоги в имени отделяются друг от друга символом-разделителем. По традиции в Windows разделителем является символ \ (обратная косая черта — backslash). В программе на C++ этот символ необходимо писать дважды `\\`. Впрочем, допускается использовать обычную косую черту `/`. Разделитель должен отделять диск от имени каталога. Имена каталогов "." (точка) и ".." (двоеточие) обозначают текущий каталог и его родительский каталог.

Для размещения наших файлов в нужных каталогах мы должны научиться перемещаться по каталогам. Так как файловая система — это составная часть операционной системы и к C++ не имеет никакого отношения, то для передвижения по каталогам необходимо использовать функции API Windows. Состав функций API все еще зависит от версии Windows, хотя Microsoft в последние годы и предприняла существенные усилия, чтобы унифицировать API. На моем домашнем компьютере установлена ОС Windows XP, которая является наследницей Windows NT, поэтому в ней работают практически все функции API Windows NT. Чтобы узнать имя текущего каталога, надо использовать функцию `GetCurrentDirectory` (ее текст приведен в листинге 6.3).

Листинг 6.3. Функция получения имени текущего каталога

```
DWORD GetCurrentDirectory(
   DWORD nBufferLength    //  размер массива символов
   LPTSTR lpBuffer        //  символьный массив для имени каталога
                        );
```

Параметры:

- `nBufferLength` — задает длину символьного массива, в который помещается имя текущего каталога;
- `lpBuffer` — символьный массив, куда помещается имя текущего каталога.

Если функция завершилась нормально, то возвращаемое значение равно длине имени-строки, помещенной в `lpBuffer`. Завершающий ноль не учитывается. Если завершение ненормальное, то функция возвращает ноль. Напишем программу, которая выводит имя текущего каталога на экран (ее текст приведен в листинге 6.4). Если в имени каталога присутствуют русские буквы, то при выводе на экран требуется перекодировка — для этого подключим наш файл Rus.h (см. листинг 5.4).

Листинг 6.4. Вывод имени каталога на экран

```
#include <iostream>
#include <Rus.h>
int main()
{  char pp[MAX_PATH]; long cc = MAX_PATH;
   DWORD t = GetCurrentDirectory(cc, pp);
   cout << Rus(pp, pp) << endl;
   return 0;
}
```

Как видите, при подключении заголовка `windows.h` мы можем использовать в программе на C++ любую константу (`MAX_PATH`) и любой тип данных (`DWORD`), который определен в Windows.

Для того чтобы иметь возможность записывать файл в нужный нам каталог, мы должны использовать функцию `SetCurrentDirectory`, текст которой приведен в листинге 6.5.

Листинг 6.5. Функция `SetCurrentDirectory`

```
BOOL SetCurrentDirectory(
  LPCTSTR lpPathName     //  новый текущий каталог
                        );
```

Параметр `lpPathName` — строковая константа или символьный массив — имя нового текущего каталога.

Напишем опять простую программу (ее текст приведен в листинге 6.6), в которой сначала сохраним текущий каталог, потом установим в качестве текущего главный каталог дискеты **a**, а потом опять установим прежний.

Листинг 6.6. Установка нового текущего каталога

```
#include <iostream>
#include <Rus.h>
int main()
{  char pp[MAX_PATH], tt[MAX_PATH];
   long cc = MAX_PATH;
   DWORD t = GetCurrentDirectory(cc, pp);      //  сохраняем
   cout << Rus(pp, tt) << endl;                //  перекодируем
   SetCurrentDirectory("a:");                  //  устанавливаем
   t = GetCurrentDirectory(cc, tt);            //  проверяем
   cout << Rus(tt, tt) << endl;
   SetCurrentDirectory(pp);                    //  возвращаем старый
   t = GetCurrentDirectory(cc, pp);            //  проверяем
   cout << Rus(pp, pp) << endl;
}
```

В этой программе нас опять подстерегают "подводные камни", которых так много в программировании. Обратите внимание — мы сохраняем текущий каталог в массиве `pp`, а для перекодировки и вывода на экран используем другой массив `tt`. Тот же самый массив использовать нельзя, потому что функция `SetCurrentDirectory` не работает с перекодированным именем. Теперь мы готовы к тому, чтобы разобраться с файловыми потоками.

Протестируемся

Сейчас повсеместно применяется тестирование, давайте и мы внесем лепту в этот процесс: напишем программу, позволяющую выяснить знания по языку C++. Для этого нам потребуется некоторое множество вопросов. Серьезные программы, как утверждают "гуру" программирования, пишутся постепенно, поэтому пусть у нас для начала будет всего 10 вопросов:

```
"Сколько типов целых чисел в C++?"
"Сколько типов дробных чисел в C++?"
"Укажите неправильный идентификатор"
"Укажите неправильное дробное число"
```

```
"Укажите неверное ключевое слово"
"Выберите правильное имя главной функции"
"Укажите неправильное имя стандартного потока"
"Укажите неверный суффикс констант"
"Сколько операторов цикла в C++?"
"Укажите оператор выхода из блока"
```

Эти десять вопросов мы запишем в файл с именем question.txt, который поместим в каталог tests на диске C:. Для этого мы должны создать каталог средствами операционной системы. Таким образом, мы сможем в дальнейшем пополнять этот файл новыми вопросами. Сначала напишем программу, а затем разберемся, как она работает (ее текст приведен в листинге 6.7).

Листинг 6.7. Программа ввода вопросов

```
#include <fstream>           //  1
#include <iostream>
#include <string>
using namespace std;
int main()
{   ofstream to("c:\\tests\\question.txt");    //  открыли и связали
   if (!to.is_open())        //  2
   { cout <<"Error! Not openfile."<<endl; return 1; }
   string s;
   while(getline(cin, s))    //  пока не <Ctrl>+<Z>
   { to << s << endl; }      //  1 вопрос на строке
   to.close();               //  3
   return 0;
}
```

Чтобы иметь возможность работать с файловыми потоками, в строке `1` мы подключили заголовок `fstream`. Это позволит нам объявлять переменные как для входных, так и для выходных потоков. В строке:

```
ofstream to("c:\\tests\\question.txt");
```

мы создаем выходной поток `to`, связываем его с нужным файлом в требуемом каталоге и одновременно открываем файл. Файла там пока еще нет, но при выполнении этой строки он там будет создан. В строке `2` с помощью метода `is_open` как раз проверяется, смогла система создать файл или нет. Если нет, то программа выдает сообщение и заканчивает работу. Закрывать файл в этом случае не требуется, т. к. он не был открыт.

Обратите внимание, что мы написали обратную косую черту дважды. Если написать один раз, то программа не находит каталог. Это довольно рас-

пространенная ошибка. Однако мы легко можем от нее избавиться, если напишем имя так:

```
c:/tests/question.txt
```

Если не указать каталог (или ошибиться в имени), то файл будет создан в текущем каталоге и на текущем диске. Если текущий каталог тот, в котором находится наша программа (например, мы зашли туда, чтобы ее запустить), то файл будет здесь же, "рядом". Мы могли бы отдельно объявить поток, а потом открыть файл:

```
ofstream to;
to.open("c:\\tests\\question.txt");
```

В данном случае это не принесет никакой выгоды, однако раздельное объявление и открытие позволяют нам связывать один и тот же поток с разными файлами. Конечно, нельзя одновременно связать одну переменную с двумя файлами, но можно это сделать по очереди, закрыв один файл и открыв другой того же типа. Объявлять новую переменную нет необходимости.

Можно делать и наоборот: один файл поочередно связывать сначала с одной переменной (например, с входным потоком), а потом — с другой (например, с выходным потоком). В данном случае мы закрываем файл в строке 3. Если выходной поток открыть и тут же закрыть, не выполняя записи, то в соответствующем каталоге будет создан пустой файл с нулевой длиной.

В цикле выполняется ввод строк и вывод их в файл. Так как в наших вопросах есть пробелы, то мы используем метод `getline`. Такой цикл, как обычно, завершается при нажатии комбинации клавиш <Ctrl>+<Z>. Вывод в файловый поток осуществляется точно так же, как и в стандартном потоке `cout`:

```
to << s << endl;
```

Манипулятор `endl` мы должны использовать для того, чтобы каждый вопрос в файле был на отдельной строке — без него все вопросы вытянутся в одну длинную строку. Перекодировать введенные таким образом строки тоже нет необходимости. Мы можем в этом убедиться, выполнив простую программу, текст которой приведен в листинге 6.8.

Листинг 6.8. Вывод файла на экран

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
int main()
```

```
{  ifstream from("c:/tests/question.txt");        //  1
   if (!from.is_open()) { cout<< "Error! Not openfile."<< endl; return 1;
}
   while (getline(from, s)) cout << s << endl;    //  2
   return 0;
}
```

В строке `1` объявлен входной поток `from`, который связывается с тем же файлом, который в предыдущей программе (см. листинг 6.7) был связан с выходным потоком. Мы не стали закрывать поток, т. к. система сама его закроет по окончании работы программы. Файл можно было открыть и в два приема: сначала перейти в нужный каталог, а потом уже открывать файл, не указывая каталог:

```
SetCurrentDirectory("c:/tests/");
ifstream from("question.txt");
```

В строке `2` мы в цикле считываем строки файла и выводим на экран. Внешне цикл ничем не отличается от соответствующего цикла в программе записи файла (см. листинг 6.7), однако остается неясным вопрос, как система узнает, что надо заканчивать читать файл. В программе записи мы явно сообщали это, нажимая комбинацию клавиш <Ctrl>+<Z>. Здесь мы ничего не сообщаем, однако система сама как-то узнает об окончании файла.

Состояния потока

Каждый поток в определенный момент времени находится в некотором состоянии. Система ввода/вывода C++ предоставляет несколько методов, с помощью которых мы всегда это состояние можем узнать [37, с. 683]:

```
bool good() const;    //  следующая операция может выполняться
bool eof() const;     //  виден конец ввода
bool fail() const;    //  следующая операция не выполняется
bool bad() const;     //  поток испорчен
```

Сами состояния определены в классе `ios_base` как константы:

```
typedef int   iostate;
  enum io_state { goodbit = 0x00,
                  badbit = 0x01,
                  eofbit = 0x02,
                  failbit = 0x04
                };
```

Как мы помним, префикс `0x` определяет шестнадцатеричную константу, однако в данном случае это не играет никакой роли, т. к. все числа меньше 10.

Если после выполнения некоторой операции поток находится в состоянии good, то это хорошая ситуация: во время предыдущей операции не произошло никаких непредвиденных событий и может быть выполнена следующая операция ввода/вывода. В остальных случаях следующая операция выполнена не будет. Состояние eof может возникнуть только при операции чтения. Для стандартного потока ввода это состояние возникает при вводе комбинации клавиш <Ctrl>+<Z>. При чтении нашего файла question.txt — после ввода последней строки. Метод getline, естественно, отслеживает это состояние. Но мы можем сделать это и в явном виде, написав соответствующее условие цикла:

```
while (!from.eof())    //  явная проверка на конец файла
{ getline(from, s); cout << s << endl; }
```

Состояния fail и bad являются состояниями ошибки потока. Если поток находится в одном из этих состояний, то операции обмена не выполняются. Состояние bad включает в себя состояние fail: когда поток находится в состоянии bad, он находится и в состоянии fail; обратное неверно. Состояние bad — это более тяжелое состояние — когда поток находится в состоянии fail, считается, что он не испорчен и никакие символы не потеряны. Когда поток в состоянии bad, ни в чем нельзя быть уверенным. Причиной ошибки может быть, например, отсутствие места на диске (дискете). Поток в состоянии fail мы можем вернуть в нормальное состояние с помощью метода clear():

```
if (stream.fail()) stream.clear();
```

После этого можно выполнять операции обмена — опять до очередной ошибки.

В базовом классе ios_base определено поле

```
iostate _M_iostate;
```

в котором сохраняются флаги состояния потока во время работы программы. Это поле мы можем прочитать методом:

```
iostate rdstate();
```

Установить любой флаг можно методом:

```
void setstate(iostate flag);
```

Макет сохранения информации

Вернемся к разработке нашей программы тестирования. На каждый из приведенных вопросов *(см. разд. "Протестируемся" данной главы)* будем предлагать 5 вариантов ответов, только один из которых будет правильный. Могут быть такие варианты ответов:

```
"1", "2", "5", "8", "10"
"1", "2", "5", "3", "7"
"a1", "_bbb", "_5_", "8_d", "abcdefgh10"
"1.0", "1e1", "1000000L", "10e-5f","-1e-1"
"for", "const", "if", "switch", "cout"
"for", "main", "cout", "function", "procedure"
"cin", "cstream", "cout", "clog", "cerr"
"A", "L", "u", "F", "f"
"1", "3", "2", "4", "5"
"return", "goto", "continue", "break", "case"
```

Номера правильных ответов следующие: `4`, `4`, `4`, `3`, `5`, `2`, `2`, `1`, `2`, `4`. Варианты ответов и правильный ответ тоже необходимо записать в файл. Так как эта информация неразрывно связана с вопросом, лучше сохранить ее в одном файле. Для этого мы должны разработать макет ввода для записи вопроса в файл. Макет зависит, в первую очередь, от того, каким образом мы будем читать информацию из файла.

Вопрос будет занимать первую строку макета. На второй строке мы разместим все остальное, отделив один элемент от другого разделителем. Очевидно, что разделителем может быть символ `$` или символ `@`, которые не входят в алфавит C++. Тогда читать один вариант ответа нам придется в символьном массиве методом `getline` (методом потока, а не методом `string`) с использованием явного разделителя. В этом случае мы можем поместить номер правильного ответа после всех вариантов. Таким образом, первый вопрос в файле может выглядеть, например, так:

```
Сколько типов целых чисел в C++?
1@2@5@8@10@4
```

Независимо от последующего чтения записывать вторую строку мы можем как одну переменную типа `string`. Кроме того, такой макет позволяет нам отслеживать элементарные ошибки набора во второй строке: разделителей должно быть ровно пять, и после пятого должно быть целое число в диапазоне от 1 до 5. Проверка данных при вводе — это очень важно, т. к. на вводе делается больше всего ошибок. Итак, наша программа ввода вопросов на данный момент выглядит так (текст программы приведен в листинге 6.9).

Листинг 6.9. Ввод вопросов для тестирования

```
#include <fstream>
#include <iostream>
using namespace std;
#include <Rus.h>
bool isRight(char digit)
```

```
{ return (('1' <= digit) && (digit <= '5')); }
int main()
{  char ss[100]; string s;
   ofstream to("c:/tests/question.txt");
   if (!to.is_open())
   { cout << Rus("Файл не открывается!", ss) << endl; return 1; }
   while (cout << Rus("Напечатайте вопрос!", ss)
               << endl, getline(cin, s))
   { to << s << endl;
   variants:    //  если ошибка, то начинаем опять отсюда
   cout << Rus("Варианты Эталон(в1@в2@в3@в4@в5@n):", ss);
   getline(cin, s);
   if (count(s.begin(), s.end(), '@') != 5)
   { cout << Rus("Вы задали не 5 вариантов!",ss) << endl;
     goto variants;
   };
   if (!isRight(s[s.size() - 1]))
   { cout << Rus("Неправильный номер эталонного ответа!", ss);
     goto variants;
   };
   to << s << endl;
   }
   to.close();
   return 0;
}
```

Основная часть программы — цикл, в котором вводятся первая и вторая строка вопроса из файла question.txt. При вводе второй строки проверяется количество вариантов ответов (по количеству разделителей) и значение последнего символа. Если выявлены ошибки, то выполняется возврат на метку `variants`, чтобы заново ввести вторую строку. В результате работы нашей программы в каталоге tests должен образоваться файл question.txt следующего содержания:

```
Сколько типов целых чисел в C++?
1@2@5@8@10@4
Сколько типов дробных чисел в C++?
1@2@5@3@7@4
Укажите неправильный идентификатор
a1@_bbb@_5_@8_d@abcdefgh10@4
Укажите неправильное дробное число
1.0@1e1@1000000L@10e-5f@-1e-1@3
Укажите неверное ключевое слово
for@const@if@switch@cout@5
```

```
Выберите правильное имя главной функции
for@main@cout@function@procedure@2
Укажите неправильное имя стандартного потока
cin@cstream@cout@clog@cerr@2
Укажите неверный суффикс констант
A@L@u@F@f@1
Сколько операторов цикла в C++?
1@3@2@4@5@2
Укажите оператор выхода из блока
return@goto@continue@break@case@4
```

Программа тестирования

Программа тестирования должна считать вопросы из файла, затем в цикле задавать очередной вопрос, предлагать варианты ответов и получать от студента номер выбранного им варианта. Эта же программа должна оценивать ответы студентов: за каждый правильный ответ студенту начисляется 1 балл. Баллы суммируются, и после прохождения всего теста выставляется оценка по следующему правилу: если правильных ответов менее 5, то оценка — 2; если правильных ответов 5 или 6, то оценка — 3; четверка выставляется за 7 или 8 правильных ответов, а 5 можно получить, если ответишь правильно на 9 или 10 вопросов. Таким образом, схема нашей программы выглядит так:

```
прочитать вопросы из файла
цикл по вопросам
  вывести вопрос; вывести варианты ответа; получить ответ
  если ответ = эталону то добавить в сумму
если сумма < 5 то вывести "неудовлетворительно"
если сумма = 5 или сумма = 6 то вывести "удовлетворительно"
если сумма = 7 или сумма = 8 то вывести "хорошо"
если сумма = 9 или сумма = 10 то вывести "отлично"
```

Для реализации программы нам необходимо решить, в какую структуру мы будем считывать вопросы из файла. Принцип инкапсуляции требует объединить все данные об одном вопросе в единую структуру, которая могла бы выглядеть так:

```
struct Questions
{  string question;     //  вопрос
   string answer[5];    //  варианты ответов
   char right;          //  номер эталонного ответа
};
```

Тогда можно использовать один из контейнеров STL, например вектор:

```
vector <Questions> Q;
```

Непосредственно считывать вопросы из файла и размещать их в элементах вектора мы не можем, т. к. определили макет с разделителями. Поэтому для ввода вариантов ответа нам потребуется символьный массив, куда мы будем считывать варианты ответов. Далее будем заносить их в массив `answer` структуры соответствующего элемента вектора. Оформим процедуру чтения как функцию (ее текст приведен в листинге 6.10), параметром которой будет наш вектор. Тем самым мы инкапсулируем все, что касается чтения из файла вопросов в одном месте, и в дальнейшем нам будет проще вносить изменения.

Листинг 6.10. Функция чтения файла вопросов

```
void ReadQuestion(vector <Questions> &Q)
{  Questions tmp;
   char s[5][30];                          //  варианты ответов
   ifstream from("c:/tests/question.txt");
   while (!from.eof())
   {  getline(from, tmp.question);         //  чтение вопроса
      for (int i = 0; i < 5; ++i)
      {  from.getline(s[i], 30, '@');      //  чтение вариантов
         tmp.answer[i].assign(s[i]);       //  присвоили в структуру
      }
      from.getline(tmp.right, 2, '\n');    //  A
      Q.push_back(tmp);                    //  прицепили к вектору
   }
}
```

Чтобы не отвлекаться от главного, мы не стали писать проверку ошибок ввода/вывода, хотя в "настоящей" системе тестирования это надо делать. В функции нужно обратить внимание на несколько важных моментов:

- мы не закрываем файл — т. к. переменная `from` является локальной, при выходе из функции система сама его закроет;
- поскольку в конце второй строки в каждом вопросе у нас стоит символ `'\n'` ("новая строка"), мы считываем правильный ответ методом `getline`. Для этого необходимо изменить определение поля `right` в структуре `Questions` на `char right[2]`, т. к. требуется место для нуля;
- при вводе правильного ответа в строке `A` мы опять прописали стандартный разделитель явным образом, т. к. иначе считывание просто не выполняется.

Замечание

Возможно, это особенность системы ввода/вывода в моей версии Visual C++ 6. Рекомендую проверить на практике в вашей системе, т. к. версии библиотеки ввода/вывода в различных системах разные, а в документации зачастую не отражены подобные "мелочи".

Наша главная программа тогда будет такой (ее текст приведен в листинге 6.11).

Листинг 6.11. Главная программа тестирования

```
int main()
{  char ss[100];       //  для перекодировки
   vector <Questions> v;
   ReadQuestion(v);    //  считываем файл в вектор
   unsigned int i = 0, summa = 0;
   int j; char a[2];
   while (i < v.size())
   {  cout << i + 1 << ". " << v[i].question << endl;
      for (j = 0; j < 5; ++j)
      cout << ' ' << j+1 << ")" << v[i].answer[j] << endl;
      cout << Rus("Выберите ответ:_", ss); cin.getline(a, 2, '\n');
      if (a[0] == v[i].right[0]) ++summa;
      ++i;
   }
   cout << Rus("Ваш результат = ", ss) << summa;
   cout << Rus(" из 10.", ss) << endl;
   if (summa < 5) cout << Rus("Неудовлетворительно", ss) << endl;
   if (summa==5 || summa==6) cout << Rus("Удовлетворительно, ss) << endl;
   if (summa==7 || summa==8) cout << Rus("Хорошо!", ss) << endl;
   if (summa==9 || summa==10) cout << Rus("Отлично!!", ss) << endl;
   return 0;
}
```

Программа проста и в точности соответствует схеме, поэтому подробно на ней останавливаться не будем. Единственный нюанс заключается в том, что мы выводим вопросы без обращения к функции перекодировки. Нам и не надо этого делать, поскольку мы вводили данные в файл вопросов из стандартного потока `cout`, и русские буквы получились сразу в нужной кодировке.

Форматирование вывода

Мы немного познакомились с понятием макета информации. Однако мы оперировали строками, а макет значительно более важную роль играет при обработке чисел. Тут нам без средств форматирования не обойтись. Например, при выводе любой ведомости на принтер, чтобы ведомость выглядела красиво и читабельно, надо все данные выравнивать по границам столбцов. Собственно, нас более чем других интересуют ответы на нижеприведенные вопросы.

1. Каким образом задавать ширину поля вывода?
2. Как можно указать выравнивание данных внутри поля?
3. Как вывести дробные числа с заданной точностью?

Все это можно сделать с помощью стандартных средств форматирования. Мы уже немного знакомы с ними — это флаги и манипуляторы. Почти в каждой программе мы использовали манипулятор `endl`, а в листинге 5.23 ширину поля вывода задавали с помощью манипулятора `setw`. Для вывода булевских значений, как мы знаем, можно применять манипулятор `boolalpha`. Этот же манипулятор можно использовать и в виде флага. Флаги форматирования определены в базовом классе библиотеки ввода/вывода `ios_base` как константы перечислимого типа в шестнадцатеричной форме:

```
typedef int fmtflags;
enum{
  left = 0x0001,          //  выравнивание влево
  right = 0x0002,         //  выравнивание вправо
  internal = 0x0004,      //  знак влево, число вправо
  dec = 0x0008,           //  вывод десятичного целого
  hex = 0x0010,           //  вывод шестнадцатеричного целого
  oct = 0x0020,           //  вывод восьмеричного целого
  fixed = 0x0040,         //  вывод дробного в виде dddd.dd
  scientific = 0x0080,    //  вывод дробного в научном виде
  boolalpha = 0x0100,     //  вывод true и false
  showbase = 0x0200,      //  вывод префиксов для целых oct и hex
  showpoint = 0x0400,     //  вывод незначащих нулей спереди
  showpos = 0x0800,       //  вывод явного + для положительных целых
  skipws = 0x1000,        //  пропускать символы-разделители
  unitbuf = 0x2000,       //  очищать буфер после каждой операции
  uppercase = 0x4000      //  E и X вместо e и x
    };
```

Кроме того, определены несколько *масок*:

```
adjustfield = left | right | internal    //  выравнивание
basefield  = dec | hex | oct             //  система счисления
floatfield = scientific | fixed          //  формат дробных
```

Флаги предназначены для задания режимов работы системы ввода/вывода. В базовом классе `ios_base` определено поле:

```
fmtflags _M_fmtflags;     //  флаги
```

в котором размещаются все флаги форматирования во время работы программы. Для установки и сброса флагов система предоставляет несколько функций-методов:

```
fmtflags flags() const;
fmtflags flags(fmtflags flags);
fmtflags setf(fmtflags flag);
fmtflags setf(fmtflags flag, fmtflags mask);
void unsetf(fmtflags mask);
```

Есть также методы, позволяющие управлять точностью выводимых чисел:

```
streamsize precision() const;
streamsize precision(streamsize newprecision);
```

и шириной поля вывода:

```
streamsize width() const;
streamsize width(streamsize newwidth);
```

Кроме того, в заголовке `iomanip` определен еще ряд манипуляторов:

```
resetiosflags(ios_base::fmtflags mask);
setiosflags(ios_base::fmtflags flag);
setprecision(int n);
setw(int n);
```

В классе `basic_ios` определены также функции:

```
char_type fill() const;
char_type fill(char_type fill);
```

позволяющие задавать символ-заполнитель (по умолчанию — пробел).

Снова о программе тестирования

Вернемся к нашей программе тестирования и подумаем о возможных направлениях ее развития. Первое, самое очевидное улучшение — возможность добавлять в файл вопросы. Когда вопросов станет много, нам потребуется из множества вопросов выбирать некоторое ограниченное количество, например, 10 или 15. Мы уже писали подобную функцию (см. листинг 5.12). Это количество должно быть параметром, который можно настраивать от запуска к запуску. Параметр естественно нужно хранить в отдельном файле настроек, который мы назовем test.ini. Для этого потребуется написать две функции: одна записывает настройки в файл, другая их считывает.

Второе улучшение — кодирование (шифрование) файла вопросов. Если файл так и будет храниться в текстовом виде, то его всегда можно будет просмотреть и/или распечатать на принтере стандартными средствами операционной системы. Тогда вся система тестирования становится абсолютно бесполезной, т. к. любой студент, заранее зная ответы на вопросы, естест-

венно, ответит на "отлично". Поэтому надо зашифровать (закодировать) в файле вопросов по крайней мере вторую строку каждого вопроса, содержащую информацию о вариантах ответа и номер правильного ответа. При чтении вопроса его придется расшифровывать. Эти два действия лучше выделить в отдельные функции — что позволит нам заменять их другими, если шифр будет раскрыт.

Примечание

Вопрос шифрования приобретает особую важность при передаче информации по сети Интернет, поэтому знание основ шифровки никогда не помешает.

Кроме ввода вопросов нужна еще функция просмотра вопросов на экране по одному. В этой функции мы постепенно сможем добавлять реализацию операций редактирования файла вопросов. Первой реализуемой операцией пусть будет операция удаления показанного на экране вопроса. Операция совершенно необходимая, поскольку вопрос или ответы к нему могут быть введены неправильно, поэтому эту запись надо удалить. Можно будет добавить также операцию замены показанного вопроса на новый. Для объединения всего этого "хозяйства" в единую программу нам потребуется главная программа-меню. Таким образом, в нашей (уже достаточно большой) программе тестирования требуется реализовать следующий набор функций:

1. Главная функция-меню.
2. Функция для обработки файла настроек.
3. Функция записи файла вопросов. Вызывает функцию кодирования.
4. Функции кодирования-декодирования.
5. Функция чтения файла вопросов в вектор. Эта функция будет использоваться в нескольких других функциях. Вызывает функцию декодирования.
6. Функция просмотра и редактирования. Вызывает функцию чтения.
7. Функция выборки конкретного теста. Вызывает функцию чтения.
8. Функция тестирования. Вызывает функцию выборки.

В будущем нам еще понадобится целая подсистема администрирования, в которой надо будет реализовать функции обработки списка пользователей: функции регистрации нового пользователя и назначение ему логина и пароля, просмотра и обработки списка пользователей, идентификации пользователя при тестировании, сохранения и просмотра результатов тестирования и т. д. А пока напишем главную функцию-меню. Само меню, очевидно, представляет собой массив строк, который задает предоставляемые программой возможности:

```
const int n = 5;
string mm[n]= { "0. Выход",
                "1. Тестирование",
                "2. Настройка",
                "3. Ввод вопросов",
                "4. Просмотр вопросов",
              };
```

С каждым пунктом меню, кроме пункта "Выход", должна быть связана соответствующая функция. После возврата из этой "рабочей" функции на экран опять надо выдавать меню. Так как в стандартном C++ нет никаких специальных средств для работы непосредственно с экраном, нам придется использовать стандартный поток `cout`. Сначала реализуем функцию полной очистки экрана. Такая функция пригодится нам во многих случаях, поэтому нам надо написать отдельную функцию. Тут мы поступим просто: выведем на экран 25 символов `\n`:

```
void ClearScreen(void)
{ cout << "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n" << flush; }
```

Манипулятор `flush` используется для принудительного освобождения буфера вывода. Функция вывода на экран меню тоже проста (ее текст приведен в листинге 6.12). Она получает параметр-вектор строк меню, выводит меню на экран и запрашивает у пользователя ввод номера пункта. Вид экрана представлен на рис. 6.2.

Листинг 6.12. Функция-меню

```
int Menu(vector<string>mm)
{  ClearScreen();    //  очистка экрана
   int k = -1; char ss[100]; string blank(29,' ');
   for (int i = 0; i < mm.size(); ++i)
   cout << Rus(blank + mm[i], ss) << endl;
   cout << endl << endl<< flush;
   cout << Rus(blank + "Выберите пункт меню:_", ss);
   cin >> k;
   return k;
}
```

Для выравнивания по центру экрана строк меню мы использовали строковую константу из 29 пробелов. Теперь можно написать главную функцию, текст которой приведен в листинге 6.13. Основу ее составляет оператор-переключатель.

Рис. 6.2. Вид меню на экране

Листинг 6.13. Главная функция-меню

```
int main()
{  const int n = 5;
   string mm[n]= { "0. Выход",
                   "1. Тестирование",
                   "2. Настройка",
                   "3. Ввод вопросов",
                   "4. Просмотр вопросов",
                 };
   vector<string>vMenu(mm, mm + n); int k = -1;
   while(k)
   {  k = Menu(vMenu);
      switch(k)
      {  case 0: break;                      //  выход
         case 1: testing(); break;           //  тестирование
         case 2: setOptions(); break;        //  настройки
         case 3: writeQuestions(); break;    //  ввод вопросов
         case 1: lookQuestions(); break;     //  просмотр вопросов
```

```
            default: cout << "Вы задали неверный пункт меню!"
        }
    }
    return 0;
}
```

Как видим, программа очень проста. Естественно, русские строки при выводе надо перекодировать. Для отладки программы нам нет необходимости иметь все вызываемые функции. Мы можем использовать функцию-заглушку, выводящую на экран сообщение, например:

```
void setOptions() { cout << "setOptions();" << endl; }
```

Теперь напишем функции записи и чтения файла вопросов.

Режимы открытия потоков (файлов)

Наша программа ввода вопросов (см. листинг 6.9) осуществляет "одноразовый" ввод. Если мы сейчас попытаемся ввести новый вопрос в файл question.txt, то окажется, что ранее записанные вопросы исчезнут. Программа записывает новый вопрос в начало файла, стирая то, что уже было записано. Это очень неудобно. Вообще-то с файлом вопросов необходимо выполнять разнообразные операции: добавлять вопросы в файл, удалять отдельные вопросы или группы вопросов, заменять вопрос и/или варианты ответов. Все эти операции являются совершенно стандартными при работе с базами данных, однако потоки C++ — это не система управления базой данных, они намного проще. Поэтому все эти операции требуется реализовать.

Памятуя о том, что программы создаются постепенно, сначала реализуем операцию добавления вопросов в файл. Это можно сделать, если открыть файл question.txt в режиме добавления — надо просто заменить строку объявления потока в программе ввода вопросов (см. листинг 6.9) на следующую:

```
ofstream to("c:/tests/question.txt", ios_base::app);
```

Вся остальная программа остается без изменения. Константа `app` определена в базовом классе `ios_base` библиотеки ввода/вывода, что и подчеркивает операция глобального доступа. В этом классе определены и другие константы, определяющие режимы открытия файла. Сам автор языка Б. Страуструп советует [37, с. 705] "покопаться" в справке и поэкспериментировать со значениями режимов открытия. Например, в справке интегрированной среды C++ Builder 6 в описании класса `ios_base` можно обнаружить такое определение этих констант (пример приведен в листинге 6.14).

Листинг 6.14. Определение констант — режимов открытия потоков

```
typedef int openmode;
enum open_mode { app = 0x01,
                 binary = 0x02,
                 in = 0x04,
                 out = 0x08,
                 trunc = 0x10,
                 ate = 0x20
               };
```

Названия констант — почти "говорящие", поэтому их назначение, за отдельными исключениями, понятно:

- `in` — открыть файл для ввода. По умолчанию в этом режиме открывается входной поток `ifstream`;
- `out` — открыть файл для вывода. По умолчанию в этом режиме открывается выходной поток `ofstream`;
- `app` — открыть для добавления. Так можно открывать либо выходной поток `ofstream`, либо двунаправленный поток `fstream`;
- `trunc` — открыть в режиме усечения. Если файл существует, то все данные в нем стираются; по умолчанию в этом режиме открывается выходной поток `ofstream`;
- `ate` — в справке открыть и немедленно перейти в конец. Отличие от `app` в том, что не предполагается немедленной дозаписи в конец файла, поэтому таким образом можно открывать и входной поток;
- `binary` — открыть в двоичном режиме.

Как предупреждал Б. Страуструп, в вашей системе могут быть и другие режимы открытия. Например, в заголовке `ios.h` системы Visual C++ 6 и в заголовке `iostream.h` системы Borland C++ 5 определены еще две константы:

- `nocreate` — открывает только существующий файл, не создавая нового, если файла нет;
- `noreplace` — наоборот, только создает новый файл; если файл существует, то он не открывается.

Мы не будем подробно на них останавливаться. Гораздо более важно разобраться с режимом `binary`. Тем более что свойство это нам понадобится для шифрования файла вопросов.

Текстовые и двоичные файлы

Текстовые файлы, с которыми мы имели дело до сих пор, отличаются следующими свойствами:

- файлы делятся на строки, и конец строки отмечается специальным символом "new line" (новая строка), роль которого в системе ввода/вывода C++ играет `'\n'`;
- при вводе чисел выполняется преобразование из символьного вида во внутренний формат (вспомните функцию `atoi` — см. листинг 3.12); при выводе чисел — из внутреннего формата в символьный.

Двоичные файлы, как вы понимаете, этими свойствами не обладают. Это означает, что двоичные файлы на строки не делятся, и при вводе/выводе никаких преобразований не делается. При операции записи в двоичный файл попадает ровно столько байт, сколько записываемый объект занимает в памяти. Например, целое число, записанное в двоичный файл, займет на диске `sizeof(int)` байт. Это существенно отличается от записи в текстовый файл, где количество записываемых по умолчанию символов зависит от величины числа. Например, число 1 в текстовом файле займет 1 байт, а 55 555 — 5 байт. Примерами двоичных файлов могут служить файлы выполняемых программ (с расширением exe). Чтобы почувствовать разницу, давайте напишем программу, которая генерируют 1000 целых случайных чисел и выводит их в файлы. Один файл будет текстовым, а другой — двоичным.

Тут нас поджидает одна проблема: в классах `fstream`, `ifstream`, `ofstream` отсутствуют методы ввода/вывода целых чисел в файл. Однако нам нет необходимости писать собственные функции, если мы умеем работать с указателями. Первый параметр методов `read`/`write` — это `char *`. Поэтому мы можем непосредственно обращаться к этим методам, используя подходящее преобразование указателей. Посмотрите, как это делается для типа `long`: `from` — входной поток, `to` — выходной поток. Текст примера приведен в листинге 6.15.

Листинг 6.15. Ввод/вывод встроенных типов данных

```
long a;
from.read((char *)&a, sizeof(long));
to.write((char *)&a, sizeof(long));
```

Преобразование типов можно делать и в стиле C++:

```
to.write(reinterpret_cast<char *>(&a), sizeof(long));
from.read(reinterpret_cast<char *>(&a), sizeof(long));
```

Точно так же можно оперировать и с любым другим типом данных. Например, мы можем вводить и выводить переменные типа `Date` так:

```
to.write(reinterpret_cast<char *>(&a), sizeof(Date));
from.read(reinterpret_cast<char *>(&a), sizeof(Date));
```

Вывод 1000 чисел принципиально ничем не отличается от вывода одного числа, однако мы все-таки напишем и выполним программу, чтобы воочию увидеть разницу в записи двоичного и текстового файла. Не будем уже писать ни функции, ни заголовки библиотек. Текст программы приведен в листинге 6.16.

Листинг 6.16. Программа вывода 1000 чисел

```
int main()
{  ofstream toBin("c:/tests/number.bin", ios::binary);
   time_t t; srand((unsigned) time(&t));
   for (int i = 0; i < 1000; ++i)
   {  int t = rand();
      toBin.write(reinterpret_cast<char *>(&t), sizeof(int));
   }
   toBin.close();
   ofstream toTxt("c:/tests/number.txt");
   srand((unsigned) time(&t));
   for (i = 0; i < 1000; ++i) toTxt << rand() << endl;
   toTxt.close();
   return 0;
}
```

Если мы несколько раз "прогоним" эту программу, и посмотрим на длину файлов, то заметим, что длина файла number.bin постоянна и всегда равна `4000` (`1000 * 4`). Длина же текстового файла number.txt меняется от запуска к запуску. Однако текстовый файл всегда можно просмотреть стандартными средствами, например, с помощью текстового редактора Notepad. Содержимое двоичного файла в этом случае представляется на экране хаотичным набором малопонятных значков.

Двоичные файлы и прямой доступ

Потоки ввода/вывода являются последовательными. Однако потоки, связанные с двоичными файлами, позволяют организовать прямой доступ к информации. Наш файл number.bin, так же, как и массив, состоит из однотипных элементов — записей, причем все записи — одинаковой длины. Смещение `k`-й записи можно вычислить по формуле `k * sizeof(тип записи)`. Библиотека ввода/вывода предоставляет средства перемещения к нужной позиции потока, связанного с файлом. При открытии потока (файла) текущая позиция равна нулю. После каждой операции чтения/записи текущая позиция изменяется в соответствии с количеством символов, участвующих в обмене. Текущую позицию всегда можно узнать — в классе `istream` определена функция-метод:

```
pos_type tellg();
```

Аналогичная функция есть и в классе ostream, только суффикс у нее другой:

```
pos_type tellp();
```

Соответственно, можно переместиться к заданной позиции с помощью методов seekg(pos_type) и seekp(pos_type), которые возвращают ссылку на поток. Эти методы имеют и относительную форму (суффикс не указан):

```
seek(offs, pos)
```

Параметр pos может принимать одно из трех значений, которые определены в ios_base как константы:

```
static const seekdir beg,    //  поиск от начала файла
                     cur,    //  поиск от текущей позиции
                     end;    //  поиск назад от конца файла
```

На сколько символов смещаться, определяется первым параметром offs. Если задается положительное число, то перемещение происходит вперед — к концу файла, если отрицательное, то назад — к началу файла. Таким образом, мы можем перезаписать любую запись двоичного файла: запись читается, затем выполняется перемещение назад на sizeof(тип записи) символов и выполняется вывод.

Шифрование файлов

Уже вывод в двоичном виде не позволяет просматривать файл простыми средствами. Однако, чтобы еще затруднить расшифровку вопросов и ответов, мы должны в программе ввода вопросов зашифровать как вопросы, так и ответы.

Способов шифрования — великое множество [41, 45]. Однако мы реализуем один из простейших, используя для этого битовую операцию исключающее ИЛИ, обозначаемую символом ^. Эта операция не имеет аналога среди логических и выполняется по таким правилам:

```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 0 = 0
```

Идея состоит в том, чтобы выполнить над каждым записываемым байтом операцию ^ с подходящей константой:

```
ch ^= const;
```

При вводе та же операция с той же константой восстановит байт в нормальное состояние. Константа может иметь значение, например, 0x55, которое в двоичной системе равно 01010101, или 0xAA, которая в двоичном виде равна 10101010.

Лучше всего оформить кодирование и декодирование в виде двух функций, текст которых приведен в листинге 6.17. Это соответствует принципу инкапсуляции и даст нам возможность менять эти функции независимо от остальных.

Листинг 6.17. Функции шифрования и расшифровывания

```
char Code(char ch) { ch ^= 0xAA; return ch; }
char Decode(char ch) { ch ^= 0xAA; return ch; }
```

Другой распространенный вариант кодирования — переставить в байте местами первую и вторую половину:

```
char Code(char ch)
{ char L = ch & 0xf0; char R = ch & 0x0f; return ((L >> 4)|(R << 4)); }
char Decode(char ch)
{ char L = ch & 0xf0; char R = ch & 0x0f; return ((L >> 4)|(R << 4)); }
```

Как видите, функции шифровки и расшифровки получаются совершенно одинаковыми. Это общий принцип. Мы написали две функции только из соображений читабельности.

Теперь можно написать функцию вывода в файл. Однако понятно, что мы не сможем работать с текстовым файлом, т. к. деления на строки не будет. Соответственно, писать и читать файл построчно с использованием разделителей мы так же не сможем. Остается использовать байтовый ввод/вывод `read`/`write` или символьный `get`/`put`. Нам придется самостоятельно формировать строки для вывода на экран. Это обычная "плата" за возможность засекречивания информации. Но чтобы это делать при вводе, нам нужно физически обозначить конец вопроса и концы ответов, т. к. символы конца строки не записываются в двоичный файл по умолчанию. Давайте в конце вопроса будем вводить `#`. Отслеживать ответы мы будем по тем же разделителям. Таким образом, ответы запишутся непосредственно после вопроса. Кроме того, при вводе номера правильного ответа в файл не будет записываться символ новой строки, поэтому при чтении файла вопросов правильный ответ может быть представлен одним символом (как и было задумано изначально), а не двумя. Здесь потребуется простая функция вывода строки, текст которой приведен в листинге 6.18.

Листинг 6.18. Функция ввода вопросов

```
void writeString(ofstream to, const string &s)
{ for(unsigned int i = 0; i < s.length(); ++i) to.put(Code(s[i])); }
void WriteQuestions(void)
{  char ss[100]; string s;
```

```
   ofstream to("c:/tests/question.bin", ios_base::app|ios_base::binary);
   if (!to) { cout << Rus("Файл не открывается!", ss) << endl; return; }
   while(cout << Rus("Напечатайте вопрос!", ss) << endl, getline(cin, s))
   {  writeString(to, s);
      variants:
      cout << Rus("Варианты эталон(в1@в2@в3@в4@в5@n:", ss);
      getline(cin, s);
      if (count(s.begin(),s.end(),'@') != 5)
      {  cout << Rus("Вы задали не 5 вариантов!", ss) << endl;
         goto variants;
      };
      if (!isRight(s[s.size() - 1]))
      {  cout << Rus("Вы задали неверный номер правильного ответа!", ss)
              << endl;
         goto variants;
      };
      writeString(to, s);
   }
   to.close();
}
```

Как видите, по сравнению с предыдущим вариантом мало что изменилось — только непосредственно вывод в файл. Тут нас подстерегает одна проблема: как проверять правильность работы функции вывода? Ведь кодирование не позволяет нам использовать программу вывода на экран (см. листинг 6.8). Надо вместо настоящей функции шифрования использовать функцию-заглушку:

```
char Code(char ch) { return ch; }
```

которая реально ничего не делает, но имеет прототип нашей функции.

Функция чтения файла вопросов "симметрична" функции вывода. Правда, она ничего не запрашивает у пользователя, а просто читает и декодирует файл вопросов. Вопросы записываются в вектор, из которого потом будет делаться выборка конкретного теста в соответствии с файлом настроек. Этот вектор функция чтения пусть получает как параметр. Текст функции ввода файла вопросов приведен в листинге 6.19.

Листинг 6.19. Функция ввода файла вопросов

```
void ReadQuestion(vector <Questions> &Q)
{  Questions tmp; char ch; string s;
   ifstream from("c:/tests/question.bin");
   if (!from) cout << "Error! Not openfile." << endl;
```

```
   from.get(ch);                  //  предварительное чтение
   while (!from.eof())
   { from.putback(ch);            //  возврат символа в поток
     s = "";
     while (ch = from.get())      //  чтение символа
     {  ch = Decode(ch);          //  декодирование
        if (ch == '#') break;     //  конец вопроса?
        s += ch;                  //  нет — прицепили символ
      };
     tmp.question=s;                 //  сделали вопрос
     for (int i = 0; i < 5; ++i)     //  вводим варианты
     {  s = "";
        while (ch = from.get())
        {  ch = Decode(ch);          //  декодировали символ
           if (ch == '@') break;     //  конец варианта?
           s += ch;                  //  нет — прицепили
        }
        tmp.answer[i] = s;           //  сделали вариант ответа
     }
   ch = from.get();      //  читаем номер ответа
   ch = Decode(ch);      //  декодировали ответ
   tmp.right = ch;       //  сделали ответ
   Q.push_back(tmp);     //  занесли в вектор
   from.get(ch);         //  опережающее чтение
   }
}
```

В этой функции неясным является только один вопрос: зачем символ сначала читается, а потом возвращается в поток? Это связано с особенностью установки признака конца файла `eofbit`: этот признак устанавливается только после операции чтения, которая выполняется после последнего байта в файле. Даже если мы прочитаем последний байт файла, `eofbit` не выставляется — это произойдет только при следующей операции чтения. Именно поэтому в конце нашего цикла, который составляет одну операцию ввода вопроса, производится опережающее чтение одного символа. Если ранее был прочитан и сформирован не последний вопрос, то нам надо вернуть прочитанный символ в поток, т. к. он входит в следующий вопрос. Если же мы обработали последний вопрос файла вопросов, то это чтение выставит бит конца файла, и цикл завершится.

Используя другую форму операции `get`, можно обойтись и без этих фокусов с возвратом символа. Посмотрите, как в этом случае выглядит главный цикл нашей функции чтения:

```
while (from.get(ch))            //  прочитали
{  s = ""; s += Decode(ch);     //  декодировали и прицепили
   while (from.get(ch))         //  читаем вопрос
   { ch = Decode(ch); if (ch == '#') break; s += ch; };
   tmp.question = s;
   for (int i = 0; i < 5; ++i)
   {  s = "";
      while (from.get(ch))      //  читаем варианты ответов
      { ch = Decode(ch); if (ch == '@') break; s += ch; }
         tmp.answer[i] = s;
   }
   from.get(ch);                //  читаем ответ
   tmp.right = Decode(ch);
   Q.push_back(tmp);
}
```

Как видите, нам нет необходимости явно проверять конец файла. Выход произойдет при очередном чтении символа.

Соберем все вместе

Функции задания и чтения режимов у нас совсем небольшие, их текст приведен в листинге 6.20. Пока в файле вопросов у нас только один параметр: размер выборки для конкретного теста из общего файла вопросов. Это целое число, поэтому мы просто записываем и считываем его из отдельного файла настроек. Практика тестирования показала, что обычно тест менее чем из 10—15 вопросов смысла не имеет. Поэтому мы зададим нижний предел, например, в 10 вопросов.

Листинг 6.20. Функции задания и чтения режимов

```
void writeInteger(ofstream& to, const int &n)
{ to.write(reinterpret_cast<char*>(&n), sizeof(int)); }
int readInteger(ofstream& from)
{ int n; from.read(reinterpret_cast<char*>(&n), sizeof(int)); return n; }
void setOptions()
{  ofstream f("c:/tests/test.ini", ios_base::binary);
   cout << "Задайте количество вопросов:_";
   int n;
   cin >> n;
   if (n < 10) { cout << "Вы задали число < 10." << endl;
                 cout << "По умолчанию считается 10." << endl;
                 n = 10;
               }
```

```
   writeInteger(f, n);
}
int getOptions(void)
{  ifstream f("c:/tests/test.ini", ios_base::binary);
   int n = readInteger(f); return n;
}
```

Функция `setOptions` вызывается у нас непосредственно из главной функции-меню, а функция `getOptions` — в функции тестирования. В функции тестирования необходимо изменить подсчет оценки и вывод результата. Теперь у нас не 10 вопросов, а столько, сколько прописано в настройках. Поэтому вместо явных цифр надо взять доли от количества вопросов конкретного теста. Лучше эту работу локализовать в отдельной функции — это даст нам возможность в дальнейшем варьировать способ вычисления оценки. Параметрами у этой функции являются количество вопросов теста и количество правильных ответов (ее текст приведен в листинге 6.21).

Листинг 6.21. Функция вычисления оценки

```
void What(int n, int summa)
{  char ss[100];
   cout << Rus("Ваш результат = ", ss) << summa;
   cout << Rus("из ", ss) << n << endl;
   if (summa <= n*0.5) cout << Rus("Неудовлетворительно,ss);
   if ((n*0.5 < summa) && (summa <= n*0.7))
   cout << Rus("Удовлетворительно", ss);
   if ((n*0.7 < summa) && (summa < n=*0.9))
   cout << Rus("Хорошо!", ss);
   if (n*0.9 < summa) cout << Rus("Отлично!!!", ss) << endl << endl;
}
```

Тогда наша функция тестирования выглядит так (листинг 6.22).

Листинг 6.22. Функция тестирования

```
void testing()
{  char ss[100]; vector <Questions> v;
   ReadQuestion(v);          //  читаем файл вопросов
   int k = getOptions();     //  чтение файла настроек
   vector <Questions> w;     //  конкретный тест
   w = GetTest(v, k);        //  выборка конкретного теста
   unsigned int i = 0, summa = 0; int j; char a;
   while(i < v.size())
   {  cout << i + 1 << "." << v[i].question << endl;
```

```
      for (j = 0; j < 5; ++j)
      cout << ' ' << j + 1 << ")" << v[i].answer[j] << endl;
      cout << Rus("Выберите ответ:_", ss);
      cin >> a;
      if (a == v[i].right) ++summa; ++i;
   }
   What(k, summa);     //  вычисляем и выводим оценку
   getch();            //  тормозим, чтобы увидеть результат
}
```

В функции тестирования мы вызываем не только функцию чтения файла настроек, но и функцию выборки конкретного теста (см. листинг 5.12). Она у нас совсем не изменилась. Только в операторе `typedef` мы должны прописать не вектор строк, а вектор наших структур:

```
typedef vector<Questions> Question;
```

Наконец, напишем функцию корректировки файла вопросов. Схема ее также проста: читаем весь файл вопросов в вектор, а затем будем выводить по одному вопросу на экран и спрашивать, не надо ли его удалить или исправить. После просмотра всех вопросов вектор выводится в файл. Не забудем, что в векторе все хранится без разделителей, поэтому при выводе вектора в файл мы должны опять "прицепить" разделители (`#` и `@`). Текст функции удаления вопросов приведен в листинге 6.23.

Листинг 6.23. Функция удаления вопросов

```
void lookQuestions()
{  vector <Questions> v; ReadQuestion(v);
   ClearScreen();    //  очистка экрана
   vector<Questions>::iterator i;
   for(i = v.begin(); i < v.end(); ++i)
   {  cout << i -> question << endl << endl << endl; // пропуск 2 строк
      cout << Rus("Удаляем <0-нет, 1-да>?_", ss);
      int a = 0; cin >> a; if (a) v.erase(i);
   }
   ofstream to("c:/tests/question.bin", ios_base::binary);
   for(int m = 0; m < v.size(); ++m)
   {  string s = v[m].question+'#'; writeString(to, s);
      for (int k = 0; k < 5; ++k)
      { s = v[m].answer[k]+ '@'; writeString(to, s); }
       s = v[m].right; writeString(to, s);
   }
   to.close()
}
```

После просмотра всего вектора выполняется запись вектора в файл.

Строковые потоки

Как пишет Страуструп [37], поток можно прикрепить не к файлу, а к строке. Таким образом, появляется возможность использовать механизм форматирования ввода/вывода для работы со строками. Все делается точно так же, как и с файлами, только надо использовать другие потоки. Система ввода/вывода предоставляет три вида строковых потоков:

```
istringstream;    //  входные потоки
ostringstream;    //  выходные потоки
stringstream;     //  двунаправленные потоки
```

Прописаны они в заголовке `sstream`. Чаще всего используются выходные строковые потоки — для формирования строки, предназначенной для вывода в файл или на экран. Напишем простую функцию, текст которой приведен в листинге 6.24, формирующую в строку комплексное число.

Листинг 6.24. Перевод комплексного числа в строковый поток

```
struct Complex { double re, im; };
string toString(const Complex &c)
{  ostringstream os;
   os << '<' << setprecision(3) << re(c) << '+'
             << setprecision(3) << im(c) << 'i>';
   return os.str();
}
```

На этом закончим наше краткое введение в систему ввода/вывода — нас ждут другие, не менее интересные места обширного "королевства C++".

ГЛАВА 7

Снова о функциях

Средства определения функций в C++ значительно более богаты, чем в других языках программирования. Некоторые особенности мы уже видели: перегрузка, параметры по умолчанию, шаблоны, указатели на функции. Однако C++ предоставляет программисту еще ряд совершенно уникальных возможностей: "левые" функции, функции с переменным числом параметров, перегрузку операций. И, как принято уже во всех современных языках программирования (кажется, кроме Basic), функции C++ могут быть рекурсивными.

"Левые" функции

Вызов функции, возвращающей значение, обычно используется в некотором выражении. Однако в C++ функция может возвратить ссылку. Такая возможность появилась исключительно в силу необходимости перегрузки операций присваивания и индексирования *(см. гл. 8)*. Однако раз уж эта возможность появилась, C++ разрешает писать независимые функции, возвращающие ссылку. Такая функция может стоять слева от знака присваивания. Рассмотрим некоторые проблемы, возникающие при ее использовании.

Уже при определении функции, возвращающей ссылку, возникает вопрос: как, собственно, ее вернуть? Это не совсем простая задача, как может показаться на первый взгляд. Следующее определение функции приводит к непредсказуемым результатам при выполнении программы:

```
int& ff(int k)
{ int d; . . .  return d; }
```

В данном определении возвращается ссылка на локальную переменную, что очень опасно. Поэтому Visual C++ 6 такую конструкцию "обложит" предупреждением, хотя и пропустит:

```
warning C4172: returning address of local variable or temporary
возвращается адрес локальной или временной переменной
```

Аналогичное предупреждение мы получим, если попытаемся возвратить параметр, передаваемый по значению. Тем не менее попробуем выполнить программу (ее тескт приведен в листинге 7.1), в которой функция, возвращающая ссылку, определена неверно.

Листинг 7.1. Неправильная функция, возвращающая ссылку

```
int n = 2;          //  глобальная переменная
int& ff(int k)      //  неправильная функция, возвращающая ссылку
{ return k; }       //  возвращаем параметр, передаваемый по значению
int f(int &i)       //  функция с параметром-ссылкой
{ return i; }
int main(void)
{  cout << f(ff(n)) << endl;  //  вызов функции на месте параметра-ссылки
   int g = 5;
   cout << f(g) <<endl;
   return 0;
}
```

Оператор

```
cout << f(ff(n)) << endl;
```

выводит на экран "мифическое" число `6684064`, что, конечно, далеко от желаемого. Второй оператор вывода, как и положено, выводит число `5`. Каким же образом решить эту проблему? Выхода, как часто бывает, два:

- передавать сам параметр по ссылке `int& ff(int& k)`;
- возвращать ссылку на статическую переменную.

Первый вариант понятен и не требует особых комментариев — очевидно, что функция может вернуть ссылку, если она ее получит в качестве параметра. Для второго случая определение функции выглядит следующим образом:

```
int& ff(int k)
{ static int d; . . . return d; }
```

Этот вариант определения функции не вызывает "комментариев" Visual C++ 6. Это и понятно — несмотря на локальную область видимости, такая переменная "живет" и после возврата из функции `f`.

Разобравшись с возможностью возврата ссылки, перейдем к основной теме — "левым" функциям. Рассмотрим элементарный пример, текст которого приведен в листинге 7.2.

Листинг 7.2. "Левая" функция выбора максимума двух чисел

```
double& max(double &x, double &y)
{ return (x>y ? x:y); }
```

Данная функция может использоваться обычным способом, например:

```
double k = max(a, b);
```

Эта же функция может быть вызвана слева от знака присваивания для того, чтобы изменить значение той переменной, которая изначально больше:

```
double a = 5, b = 6, c = 7;
max(a, b) = 10;
```

Такая запись приведет к тому, что значение переменной `b` станет равным `10`. Конечно, это проще всем привычной записи:

```
if (a > b) a = 10; else b = 10.
```

Возврат ссылки обеспечивает и более интересные выражения, например:

```
max(max(a, b), c) = 10;         //  c = 10
max(max(a, b) = 10, c) = 0;     //  b = 0;
```

В первой строке двукратный вызов обеспечивает присвоение максимальной из трех переменных (это переменная `c`) значения `10`. Во второй строке сначала максимуму из `a` и `b` присваивается `10`, затем это число сравнивается с переменной `c`, равной `7`, и максимум (переменная `b`) обнуляется.

Эта возможность особенно полезна при обработке массивов. Допустим, нам надо отыскать в массиве минимальный элемент и обнулить его. Обычным приемом в таких случаях является вычисление индексов, и затем присвоение нуля элементу с данными индексами. "Левая" функция, возвращающая ссылку на элемент массива, приведена в листинге 7.3.

Листинг 7.3. "Левая" функция для максимума массива

```
int& Mmin(int n, int d[])
{  int im = 0;
   for (int i = 1; i < n; i++) im = d[im] < d[i] ? im : i;
   return d[im];
}
```

Как видим, реализация поиска минимального элемента довольно тривиальна и не содержит ничего необычного. Вызов справа тоже не отличается от обычного:

```
int x[] = { 10, 20, 40, 5, 6, 7, 8, 9, 50, 12 };
int m = Mmin(10, x);
```

Переменная m получит значение 5 в точном соответствии с алгоритмом функции Mmin. Но вызов такой функции слева приведет к тому, что минимальный элемент будет изменен. Например:

```
Mmin(10, x) = 0;
```

Минимальный элемент массива x, равный 5, станет равным нулю.

"Левые" функции с указателями

В *гл. 5* мы уже упоминали, что функции, возвращающие итератор, можно писать слева от знака присваивания. Однако аналогичный фокус можно проделать и с указателями. Все дело — в возврате адреса. Текст функции, возвращающей указатель на максимум из двух чисел, приведен в листинге 7.4.

Листинг 7.4. "Левая" функция максимума с указателем

```
double* max(double *x, double *y)
{ return (*x > *y? x: y); }
```

И вызов практически ничем не отличается:

```
double a = 5, b = 6;
*max(&a, &b) = 10;
```

Звездочка слева прописана в полном соответствии с семантикой присвоения значения по указателю. Если функция используется справа от присваивания, то звездочку тоже надо писать. Аналогичная функция Mmin с возвратом указателя принимает такой вид:

```
int* Mmin(int n, int d[])
{  int im = 0;
   for(int i = 1; i < n; i++) im = d[im] < d[i]? im : i;
   return &d[im];
}
```

Вызовы функции также прописываются со звездочкой:

```
int x[] = { 10, 20, 40, 5, 6, 7, 8, 9, 50, 60 };
int m = *Mmin(10, x);
*Mmin(10, x) = 0;
```

Поскольку и указатели, и ссылки представляют собой фактически адреса, то возникает естественный вопрос: а нельзя ли вместо ссылки вернуть указатель и, наоборот, вместо указателя — ссылку? Исследование вариантов функции max в Visual C++ 6 для различных сочетаний указателей и ссылок

показало, что правильными являются следующие определения функции (их текст приведен в листинге 7.5).

Листинг 7.5. Смешанные "левые" определения функции

```
double* max(double &x, double &y)
{ return (x > y? &x: &y); }
double& max(double *x, double *y)
{ return (*x > *y? *x: *y); }
```

Как мы видим, в первом случае функция получает параметры-ссылки. Однако для возврата указателя мы вынуждены прописывать явное адресное выражение `&x` или `&y`. Вызывать данную функцию надо так:

```
cout << *max(a, b) << endl;
*max(a, b) =  20;
```

Вызов `max(a, b)` без звездочки выдает в качестве результата адрес наибольшего из параметров, поэтому такой вызов можно указывать в адресных выражениях.

Во втором случае, при получении параметров-указателей, в теле функции прописываются звездочки — без звездочек Visual C++ 6 выдает сообщение:

```
error C2440: 'return' : cannot convert from 'double *' to 'double &'
'return': невозможно преобразовать 'double *' в 'double &'
```

Такую функцию тоже можно вызывать и слева и справа:

```
cout << max(&a, &b) << endl;
max(&a, &b) = 10;
```

Интересно, что C++ позволяет запретить использовать функцию как "левую". Для функции, возвращающей ссылку, достаточно прописать `const` перед возвращаемым значением. Текст примера приведен в листинге 7.6.

Листинг 7.6. Запрет "левости" функции

```
const double& max(double &x, double &y)
{ return (x > y? x: y); }
```

Попытка написать вызов такой функции слева от операции присваивания приведет к ошибке трансляции:

```
error C2166: l-value specifies const object
l-value определяет константный объект
```

Для функции, возвращающей указатель, тоже можно прописать `const`, только надо учитывать, что это слово может задаваться либо один раз, либо дважды и относиться как к указателю, так и к объекту. Вариант

```
double * const max(double *x, double *y)
{ return (*x > *y? x: y); }
```

транслируется без ошибок. Это и понятно — последнее определение означает возврат константы-указателя на `double`. При вызове

```
*max(&a, &b) = 10;
```

мы изменяем значение, а не указатель, поэтому и не возникает сообщений об ошибках.

Функции с переменным числом параметров

Язык C++ вслед за C позволяет писать функции с переменным числом параметров. Одним из простых примеров может служить функция, вычисляющая среднее арифметическое своих аргументов. Другой, уже классический пример — функция сцепления произвольного количества строк, которая является естественным обобщением функции сцепления двух строк.

Переменный список параметров задается в заголовке функции многоточием:

```
int f(...)
```

Этот заголовок не вызывает у компилятора протестов. Такая запись означает, что при определении функции компилятору неизвестны ни количество параметров, ни их типы, и он, естественно, не может ничего проверить. Количество параметров и их типы становятся известными только при вызове функции.

Однако у программиста с написанием таких функций сразу возникают проблемы. Ведь имена параметров отсутствуют. Поэтому доступ можно осуществить только одним способом — косвенным, используя указатель. Вспомним, что все параметры при вызове помещаются в стек. Если мы каким-то образом установим указатель на начало списка параметров в стеке, то, манипулируя с указателем, мы, в принципе, можем "достать" все параметры.

Таким образом, список параметров совсем пустой быть не может, должен быть прописан хотя бы один явный параметр, адрес которого можно получить при выполнении программы. Заголовок функции может выглядеть так:

```
int f(int k...)
```

Ни запятая, ни пробел после параметра не обязательны, хотя можно их и прописать.

Есть одно обстоятельство, которое ограничивает применение таких функций: при написании функции с переменным числом параметров, помимо алгоритма обработки, программист должен разрабатывать и алгоритм доступа к параметрам. Так что список необъявленных параметров не может быть совсем уж произвольным — в языке C++ не существует универсальных средств распознавания элементов этого списка. Это же означает, что передача аргумента не того типа, который задумывался, или не тем способом, который подразумевался при разработке, приведет к катастрофическим последствиям — компилятор-то ничего не проверяет.

Попробуем написать функцию, вычисляющую среднее арифметическое своих аргументов. Для этого нужно разрешить несколько вопросов:

- как установить список параметров в стеке;
- как "перебирать" параметры;
- как закончить перебор.

Для доступа к списку параметров нам потребуется указатель, значением которого будет адрес последнего явного параметра в списке. Ответ на второй вопрос очевиден — надо изменять значение этого указателя, чтобы переместиться на следующий параметр. Отсюда следует, что указатель должен быть типизированным, поскольку с бестиповым указателем нельзя выполнять арифметические операции. Это же означает, что программист при разработке функции с переменным числом параметров должен отчетливо себе представлять типы аргументов, которые будет обрабатывать функция. Кроме того, способ передачи параметров должен быть одинаковым для всех параметров: либо все — по значению, либо все — по ссылке, либо все — по указателю.

Ответ на последний вопрос не вызывает затруднений. Это можно сделать одним из двух способов:

- явно передать среди обязательных параметров количество аргументов;
- добавить в конец списка аргумент с уникальным значением, по которому будет определяться конец списка параметров.

И тот и другой способ имеют право на жизнь — все определяется потребностями задачи и "вкусами" программиста. В данном случае сначала попробуем второй способ: последним значением списка параметров будет ноль. Текст функции приведен в листинге 7.7.

Листинг 7.7. Вычисление среднего арифметического аргументов (ноль в конце)

```
double f(double n, ...)     //  заголовок с переменным числом параметров
{  double *p = &n;          //  установились на начало списка параметров
   double sum = 0, count = 0;
   while (*p)               //  пока аргумент не равен нулю
```

```
   {  sum += (*p);          //  суммируем аргумент
      p++;                  //  перемещаемся на следующий аргумент
      count++;              //  считаем количество аргументов
   }
   return ((sum)?sum/count:0);     //  вычисляем среднее
}
```

Вызов функции может выглядеть так:

```
double y = f(1.0, 2.0, 3.0, 4.0, 0.0);
```

Переменная `y` получит значение `2.5`. Так как компилятор ничего не проверяет, то попытка вызвать такую функцию с целыми аргументами `f(1, 2, 3, 0)` либо вызовет аварийную остановку программы (это лучший вариант), либо приведет к неверному (но правдоподобному — в этом главная опасность) результату.

Реализация функции, которая в качестве первого параметра получает количество аргументов, на первый взгляд, не вызывает затруднений. Однако если первый аргумент — целое число, то требуется преобразование указателя. И тут не все варианты проходят. Не будет работать такой вариант:

```
double f(int n, ...)          //  количество элементов
{  int *p = &n;               //  указатель — "целый"
   double sum = 0, count = n;
   for (;n--;(double*)p++)    //  преобразование int* -> double*
   sum += (*p);
   return ((sum)?sum/count:0);
}
```

Такой вариант тоже неработоспособен:

```
double f(int n, ...)          //  количество элементов
{  double *p = (double *)&n;  //  преобразование адреса
   double sum = 0, count = n;
   for (;n--;p++)             //  изменение указателя
   sum += (*p);
   return ((sum)?sum/count:0);
}
```

Причина кроется в том, что изменение указателя производится на столько байт, сколько в памяти занимает базовый тип. В обоих случаях мы установились не на начало списка `double`-параметров, а на `sizeof(int)` байт "раньше" — на целую переменную. И от этого адреса происходит изменение указателя на 8 байт `(sizeof(double))`, что приводит к совершенно неверным результатам. Решение заключается в том, чтобы сначала изменить "целый" указатель, а потом уже его преобразовать в `double *`. Так всегда необходимо

делать, если тип первого параметра отличается от типов отсутствующих параметров. Текст правильно реализованной функции приведен в листинге 7.8.

Листинг 7.8. Вычисление среднего арифметического аргументов (количество)

```
double f(int n, ...)     //  количество элементов
{  int *p = &n;
   p++;                  //  A (установка "целого" на double)
   double *pp = (double *)p;    //  преобразование типа указателя
   double sum = 0, count = n;
   for (;n--;pp++)       //  правильное увеличение на 8
   sum += (*pp);
   return ((sum)?sum/count:0);
}
```

В строке A операция p++ устанавливает указатель на первый элемент списка параметров типа double. Для дальнейшего изменения указателя на 8 мы использовали преобразование типа указателя:

```
double *pp = (double *)p;
```

После этой строки операция pp++ будет увеличивать указатель на sizeof(double) равный 8 байт, что нам и требуется.

Мы использовали способ передачи параметров по значению. В этом случае в качестве фактических аргументов можно задавать произвольные выражения. Однако можно использовать и передачу ссылки — это несколько осложняет вызов, поскольку тогда в списке аргументов могут прописываться только переменные. Необходимо также помнить, что все фактические аргументы должны передаваться одинаковым способом. Прототип первого варианта функции выглядит так (тело функции не изменяется):

```
double f(double &n, ...)
```

При вызове можно использовать элементы массива:

```
double m[] = { 1.0, 2.0, 3.0, 4.0, 0.0 };
cout << f(m[0], m[1], m[2], m[3], m[4]) << endl;
```

Программа выведет на экран 2.5.

Язык C++ в качестве элементов переменного списка аргументов разрешает прописывать указатели. Однако обработка такого варианта вызывает сложности — нам требуется двойной косвенный доступ, а указатель для доступа к стеку — "одноразовый". Таким образом, если передавать параметры-указатели, то в приведенной программе (см. листинг 7.8) (с первым параметром-количеством) значение *pp — это не число типа double, а адрес этого числа. Тут без "обмана" компилятора не обойтись, поскольку он просто

так не пропускает преобразование "одноразовой" косвенности в двойную. Но мы помним, что все типизированные указатели, независимо от типа и косвенности всегда представляют собой адрес, размер которого в процессоре Intel — 4 байта. Поэтому для "обмана" компилятора можно использовать объединение `union`. Текст функции, реализующей этот прием, приведен в листинге 7.9.

Листинг 7.9. Переменный список параметров-указателей (количество)

```
double f(int n, ...)
{  int *p = &n;    //  "одноразовый" указатель
   p++;            //  "достаем" список параметров-указателей
   union Pointer
   { double **pp; double *kp; };    //  "подстава" указателей
   Pointer A;
   A.kp = (double *)p;              //  "обманываем" компилятор
   double sum = 0, count = n;
   for (;n--;A.pp++)                 //  изменяем двойной указатель
   sum += (**A.pp);                  //  двойной доступ
   return ((sum)?sum/count:0);
}
```

Хотя по стандарту такое использование `union` означает undefined behaviour (неопределенное поведение), и непереносимо, но на практике (например, на платформе Intel) работает хорошо. Однако при переносе на другую платформу надо будет проверять корректность работы такой функции.

Необходимо обратить внимание на то, что изменяем мы "двойной" указатель — "одноразовый" применять нельзя, поскольку будет изменение на `sizeof(double)`, равное 8 байт, а нам требуется изменение на `sizeof(double*)`, равное 4 байта. И при суммировании используется двойной косвенный доступ. Обращение к функции выполняется так:

```
cout << f(2, &a, &b) << endl;
```

Другой вариант функции со списком указателей переменной длины (без первого параметра-счетчика аргументов) может использовать в качестве признака окончания списка параметров нулевой указатель. Но в этом случае "обманывать" компилятор не требуется — в функции, текст которой приведен в листинге 7.10, непосредственно используется "двойной" указатель.

Листинг 7.10. Переменный список параметров-указателей (ноль в конце)

```
double f(double *a, ...)
{  double **p = &a;      //  берем адрес-адреса
```

```
   double sum = 0, count = 0;
   while (*p != 0)        //  NULL — прямо в списке параметров
   {  sum += (**p);       //  выбираем значения
      count++; p++;       //  "бежим" по списку
   }
   return ((sum)?sum/count:0);
}
```

Вызов функции выглядит так:

```
f(&a, &b, 0)
```

Особо обратите внимание на следующее: в списке параметров `0` — это значение, а не адрес. Поэтому в теле функции проверка окончания цикла делается с одной звездочкой, а не с двумя.

Напоследок осталось рассмотреть пример, в котором список указателей переменной длины составляют указатели на `char`. Мы выделяем этот случай по двум причинам:

- размер данных (`char`) меньше, чем размер указателя (`char *`) — в остальных случаях размер данных больше или равен размеру указателя;
- `char *` — это единственный указатель, вместо которого при вызове можно задавать не адрес.

Типичной функцией, в которой можно применить переменный список параметров, является функция сцепления произвольного количества строк в одну. Ее заголовок может выглядеть так:

```
char *f(char *s1, ...)
```

Функция должна сначала вычислить количество памяти, необходимой для целевой строки, а потом уже помещать туда результат сцепления. Используем тот же прием, что и в предыдущем примере — последний параметр должен быть `0`. Текст функции сцепления строк приведен в листинге 7.11.

Листинг 7.11. Сцепление строк (ноль в конце)

```
char *f(char *s1, ...)
{  char **cp = &s1;                 //  адрес первого указателя
   int len = 0;
    //  цикл для определения общей длины сцепляемых строк
   while (*cp) { len += strlen(*cp); cp++; }
   char *s = new char[len + 1];     //  память для строки
   s[0] = 0                         //  "очищаем" строку
    // цикл для сцепления строк
   cp = &s1;                        //  опять установка на 1-й параметр
```

```
   while (*cp)
   {  strcat(s, *cp);              //   прицепляем первую (и следующие)
      cp++;                        //   перемещаемся на следующую
   }
   return s;
}
```

Вызов функции может быть таким:

```
char *ss = f(s1, s2, s3, 0);
```

где `s1`, `s2`, `s3` — это либо объявленные константы, либо переменные типа `char *`. Ту же функцию можно вызывать и с явно прописанными константами:

```
char *sd = f("First", "Two", "Three", 0);
```

Очевидно, вместо параметра-указателя (ссылки) можно подставлять выражение, имеющее результатом указатель (ссылку). В частности, на месте указателя на `char` можно вызвать функцию, которая вводит строку с клавиатуры.

Стандартные средства

В стандарт языка входит набор макросов для работы со списками параметров переменной длины, определенный в библиотеке stdarg.h. При их использовании точно так же требуется указывать в списке явный параметр, объявить и установить на него указатель и перемещаться по списку, изменяя его. В конце списка должен стоять NULL. Макросы, обеспечивающие стандартный доступ к спискам параметров переменной длины, имеют следующие форматы:

```
void va_start(va_list prm, последний явный параметр);
тип va_arg(va_list prm, тип);
void va_end(va_list prm);
```

Тип указателя определяется с помощью оператора `typedef` как `va_list`. Макрос `va_start` устанавливает указатель типа `va_list` на явный параметр, макрос `va_arg` перемещает указатель на следующий параметр, а макрос `va_end` обнуляет указатель. Указанные макросы используются следующим образом.

1. В теле функции с переменным числом параметров до первого использования указанных макросов должно появиться объявление объекта типа `va_list`, например `va_list LastP`; фактически это является объявлением указателя.
2. Указанный объект связывается с последним явным параметром (перед многоточием) переменного списка параметров с помощью макроса

va_start, например va_start(LastP, P); таким образом происходит инициализация указателя.

3. Передвижение по переменному списку параметров выполняется макросом va_arg. Для этого необходимо явно указывать тип очередного параметра, т. е. программист должен его знать в момент написания программы. Если все параметры в списке целого типа, то вызов va_arg выглядит так: va_arg(LastP, int).
4. После всей обработки ставится вызов va_end, например va_end(LastP).

В качестве примера рассмотрим реализацию функции вычисления среднего арифметического (вариант с количеством аргументов) с использованием этих макросов. Текст примера приведен в листинге 7.12.

Листинг 7.12. Вычисление среднего с использованием стандартных средств (количество)

```
double f(int n, double a, ...)
{  va_list p;                        //  объявление указателя
   double sum = 0, count = 0;
   va_start(p, n);                   //  инициализация указателя
   while(n--)
   {  sum += va_arg(p, double);      //  перемещение указателя
      count++;
   }
   va_end(p);                        //  "закрытие" указателя
   return ((sum)?sum/count:0);
}
```

Очень похоже выглядит вариант функции с нулем в конце списка, текст которой приведен в листинге 7.13.

Листинг 7.13. Вычисление среднего стандартными средствами (ноль в конце)

```
double f(double a, ...)
{  va_list p;                         //  объявление указателя
   double sum = 0, count = 0;
   va_start(p, a);                    //  инициализация указателя
   double k = a;                      //  промежуточная переменная
   do { sum += k; count++; }
   while(k = va_arg(p, double));      //  пока не ноль, то передвигаемся
   va_end(p);                         //  "закрыли" указатель
   return ((sum)? sum/count: 0);
}
```

Однако в этом случае удобно использовать цикл `do...while`, т. к. указатель `p` сразу устанавливается на слагаемое. Передвижение по списку выполняется прямо в условии цикла, что обеспечивает одновременную проверку на ноль.

Рекурсивные функции

Язык C++ предоставляет возможность написания рекурсивных функций, однако целесообразность использования рекурсии оставляется на усмотрение программиста. Как правило, рекурсивные алгоритмы применяются там, где имеется явное рекурсивное определение обрабатываемых данных. Не будем отступать от традиции и рассмотрим функцию факториала n!. Как правило, в программировании ее определяют как произведение первых n целых чисел:

```
n! = 1 * 2 * 3 * ... * n
```

Такое произведение можно легко вычислить с помощью итеративных конструкций, например, оператора цикла `for`. Текст функции для вычисления факториала приведен в листинге 7.14.

Листинг 7.14. Итеративная функция вычисления факториала

```
long Fact(int k)
{  long f; int i;
   for (f = 1, i = 1; i < k; i++) f *= i;
   return f;
}
```

Однако существует также другое (математическое) определение факториала, в котором используется рекуррентная формула:

- $0! = 1$
- $\forall\ n > 0\ n! = n \times (n - 1)!$

Если для факториала первое (итеративное) определение может показаться проще, то для чисел Фибоначчи рекурсивное определение:

- $F(1) = 1$
- $F(2) = 1$
- $\forall\ n > 2 F(n) = F(n - 1) + F(n - 2)$

выглядит для вычислений гораздо лучше, чем прямая формула.

Понятно, что организовать вычисления по рекуррентным формулам можно и без использования рекурсии. Однако, как видно на примерах, представленных в [9, 30, 39], преобразование естественной рекурсивной формы в итеративную — довольно сложная задача. Использование рекурсии позво-

ляет легко (почти автоматически) запрограммировать вычисления по рекуррентным формулам. Например, рекурсивная функция для вычисления факториала n! имеет следующий вид (ее текст приведен в листинге 7.15).

Листинг 7.15. Рекурсивная функция вычисления факториала

```
long Fact(int k)
{  if (k == 0) return 1;
   return (k * Fact(k – 1));              //   рекурсивный вызов
}
```

Аналогично, по указанному определению легко написать функцию вычисления чисел Фибоначчи, текст которой приведен в листинге 7.16.

Листинг 7.16. Рекурсивная функция вычисления чисел Фибоначчи

```
long Fibo(int k)
{  if ((k == 2) || (k == 1)) return 1;
   return (Fibo(k – 1) + Fibo(k – 2));    //  рекурсивный вызов
}
```

Рекурсивной функцией называется функция, вызывающая саму себя в своем теле. Необходимо еще раз подчеркнуть, что "самовызов" будет рекурсивным только в том случае, если находится в теле функции. Как мы видели ранее, "самовызов" в списке параметров не является рекурсивным.

Обычно различают прямую и косвенную рекурсию. Если в теле функции явно используется вызов той же самой функции, то имеет место *прямая рекурсия*, как в приведенных примерах. Если две или более функций взаимно вызывают друг друга, то имеет место *косвенная рекурсия*. Обычно косвенная рекурсия возникает при реализации программ синтаксического анализа методом рекурсивного спуска.

Прекрасным примером рекурсивной функции является быстрая сортировка Хоара. Сама схема алгоритма является рекурсивной, поэтому написать итеративную программу достаточно сложно, что можно увидеть в книге Н. Вирта [9] и в [30, 39]. Схема функции выглядит так:

```
void QuickSort(A, 1, n)
{  //  Выбрать разделяющий элемент с номером 1 < k < n
   //  Разделить массив A относительно k-го элемента
   QuickSort(A, 1, k-1);    //  рекурсивный вызов с левой частью
   QuickSort(A, k+1, n);    //  рекурсивный вызов с правой частью
}
```

Есть большое количество традиционных "игрушечных" задач (ханойские башни, расстановка ферзей и т. п.), которые анализируются в литературе

[9, 34] для демонстрации рекурсии. Мы не будем на них останавливаться — гораздо интереснее рассмотреть типично итеративные алгоритмы, которые можно представить рекурсивным образом. В принципе, любой цикл можно заменить эквивалентной рекурсивной программой. В качестве примера, текст которого приведен в листинге 7.17, рассмотрим рекурсивную реализацию функции вычисления длины строки (см. листинг 3.11).

Листинг 7.17. Рекурсивная функция вычисления длины строки

```
unsigned int Length (char *s)
{  if (*s == 0) return 0;    //  можно просто !(*s)
   else return 1 + Len(s + 1);
}
```

Вызов такой функции абсолютно ничем не отличается от вызова аналогичной итеративной, например:

```
cout << Len("1234567890");
```

на экран совершенно правильно выводится `10`. Работа рекурсивных функций обычно имеет несколько "мистический" оттенок, поэтому не будем пока в деталях разбирать работу этой функции, но обратим внимание на несколько важных моментов:

- цикла в функции нет — вместо него у нас имеется рекурсивный вызов. Таким образом, явное повторение заменяется неявным — рекурсивным;
- параметр, который является указателем, в рекурсивном вызове увеличивается, перемещаясь к следующему символу;
- чтобы такое увеличение не происходило до бесконечности, в наличии имеется условие окончания рекурсии — в операторе `if` проверяется достижение конца строки.

Эту же функцию можно написать несколько иначе:

```
unsigned int Length (char *s)
{  if (*s) return 1 + Len(s + 1);
   else return 0;
}
```

Здесь мы наблюдаем аналогичные особенности:

- вместо цикла имеется рекурсивный вызов;
- параметр в рекурсивном вызове изменяется;
- условие окончания заменено условием продолжения.

Прежде чем делать выводы, рассмотрим еще один пример — последовательную обработку файла. Пусть открытие и закрытие файла выполняются

в главной программе, а обработка — в отдельной функции, которая последовательно читает записи файла и обрабатывает их. Традиционно такая обработка выполняется в цикле следующего вида:

```
while (не конец файла)
{  //  читать запись
   //  обработать запись
}
```

Попробуем написать рекурсивную функцию без использования цикла. Пусть потоковая переменная имеет имя `f`. Файл представляет собой файл строк — исходный текст самой этой программы, который находится в том же каталоге, что и исполняемая программа. Текст рекурсивной функции приведен в листинге 7.18.

Листинг 7.18. Рекурсивная функция обработки текстового файла

```
void ReadFile(ifstream &f)
{  char s[100];       //  буфер для строки
   getline(f, s);     //  читаем строку
   cout << s;         //  обработка строки
   if (!f.eof())      //  пока не конец файла
   ReadFile(f);       //  рекурсивный вызов
}
int main(void)
{  ifstream f("recurs.cpp");
   ReadFile(f);
   f.close();
   return 0;
}
```

Как мы видим, функция `ReadFile` несколько отличается от приведенных ранее функций — отсутствует явное изменение параметра. Параметр-то все равно изменяется, но неявно — как состояние потока `f` при чтении. Как и в предыдущих случаях, цикл заменен рекурсивным вызовом, и присутствует условие продолжения.

Формы рекурсивных функций

В общем случае любая рекурсивная функция `Rec` включает в себя некоторое множество операторов `S` и один или несколько операторов рекурсивного вызова `Rec`. Как мы видели на примерах, рекурсивный вызов обязательно сопровождался некоторым условием: либо условием окончания, либо продолжения. Это понятно, поскольку безусловные рекурсивные вызовы при-

водят к бесконечным процессам — они сродни бесконечному циклу. Вспомните детский стишок "У попа была собака..." — это как раз случай бесконечной рекурсии. Если реализовать вывод этого стишка как рекурсивную функцию, то она может выглядеть так:

```
void PriestAndDog(void)
{  cout << "У попа была собака, он ее любил.";
   cout << "Она съела кусок мяса, он ее убил,";
   cout << "вырыл яму, закопал и на камне написал:";
   PriestAndDog();
}
```

Теоретически работа этой функции никогда не завершится. На практике программа закончится аварийно по причине переполнения стека (кто хочет — может проверить). Следовательно, главное требование к рекурсивным функциям заключается в том, что вызов рекурсивной функции должен выполняться по условию. Это условие прописывается в операторе `if` и может быть либо условием продолжения, либо условием окончания. В первом случае в `if` пишется рекурсивный вызов, во втором — оператор возврата.

Примечание

Это не гарантирует завершение рекурсивной функции — можно совершить другие ошибки. Например, если в определении функции вычисления факториала рекурсивный вызов записан в форме `Fact(k--)`, то при выполнении возникает ошибка переполнения стека при любом способе передачи параметра. В этом случае при рекурсивном вызове берется значение `k` до уменьшения. В результате происходит вызов с одним и тем же значением параметра, условие никогда не удовлетворяется и рекурсия превращается в бесконечную.

Структура рекурсивной функции может принимать три разных формы (используем условие продолжения).

1. Форма с выполнением действий до рекурсивного вызова (на рекурсивном спуске): `void Rec(void) { S; if (условие) Rec(); }`.
2. Форма с выполнением действий после рекурсивного вызова (на рекурсивном возврате — подъеме): `void Rec(void) { if (условие) Rec(); S; }`.
3. Форма с выполнением действий как до (на рекурсивном спуске), так и после рекурсивного вызова (на рекурсивном возврате): `void Rec(void) { S1; if (условие) Rec(); S2; }`.

В образцах указан заголовок

```
void Rec(void)
```

поскольку в данном случае нам важно было продемонстрировать именно формы рекурсивных функций, не отвлекаясь на другие детали. Сочетания "рекурсивный спуск" и "рекурсивный подъем" связаны с понятием *глубины*

рекурсии — функция спускается на глубину рекурсии и поднимается "оттуда".

Функция вычисления факториала написана в первой форме. Нам удалось написать вторую форму функции, вычисляющей длину строки. Аналогично, поменяв условие, можно преобразовать и функцию вычисления факториала (ее текст приведен в листинге 7.19).

Листинг 7.19. Вторая форма рекурсивной функции вычисления факториала

```
long Fact(int k)
{  if (k > 0) return (k * Fact(k – 1));     //  рекурсивный вызов
   else return 1;
}
```

Однако переделать функцию чтения файла нам так просто не удастся. Попытка в "лоб" не приносит успеха:

```
void ReadFile(ifstream &f)
{  char s[100];       //  буфер для строки
   if (!f.eof())      //  пока не конец файла
   ReadFile(f);       //  рекурсивный вызов
   getline(f, s);     //  читаем строку
   cout << s;         //  обработка строки
}
```

Функция становится бесконечной. Конец файла никогда не достигается, поскольку до рекурсивного вызова не выполняется операция чтения. Это наводит на мысль о возможности преобразования данной функции в третью форму — чтение строки выполнять до проверки условия, а вывод строки — после рекурсивного вызова. Попробуем и посмотрим, что получится:

```
void ReadFile(ifstream &f)
{  char s[100];       //  буфер для строки
   getline(f, s);     //  читаем строку
   if (!f.eof())      //  пока не конец файла
   ReadFile(f);       //  рекурсивный вызов
   cout << s;         //  обработка строки
 }
```

Функция работает, но совсем не так, как ожидалось. Строки файла выводятся на экран в обратном порядке. Таким образом, для некоторой рекурсивной задачи подходит отнюдь не любая форма рекурсивной функции. Можно сказать, что для всякой рекурсивной задачи существует естественная для нее форма рекурсивной функции.

Выполнение рекурсивных функций

Несмотря на то, что запись рекурсивных функций бывает очень короткая, они часто "страдают" неэффективностью. Вспомним, что при каждом вызове в стеке должны быть размещены все параметры и локальные переменные. На эту работу (так же, как и на последующее освобождение) расходуется и время, и пространство — в программу вставляются соответствующие команды, которые выполняются при каждом вызове. Однако эта неэффективность не очень существенна по сравнению с другой — расходом памяти под стек. Поскольку рекурсивный вызов выполняется до окончания выполнения функции — размер стека увеличивается при каждом вызове до тех пор, пока не будет достигнута точка, когда выполняется возврат. Размер стека пропорционален глубине рекурсии. *Глубина рекурсии* — это максимальная степень вложенности рекурсивных вызовов. В общем случае глубина будет зависеть от входных данных. Например, в рекурсивной функции `ReadFile` глубина рекурсии определяется количеством строк в читаемом файле. В принципе, это может привести к переполнению стека и аварийному завершению программы. Поэтому при разработке рекурсивных функций необходимо минимизировать количество и размеры локальных переменных и параметров.

Существует также и другая неэффективность — лишние вычисления. В принципе, одним из методов борьбы с такой неэффективностью является сокращение количества рекурсивных обращений в тексте функции: например, если есть такая возможность, вместо двух вызовов оставить только один.

Рассмотрим еще несколько задач, реализуемых посредством рекурсивных функций, и разберемся более детально в их работе. Пусть требуется написать функцию, определяющую, является ли строка палиндромом (одинаково читающаяся слева направо и наоборот). Написание итеративного варианта не представляет сложностей:

```
bool Palindrom(string s)
{  int i = 0, j = s.length() - 1;
   for (;i < j; i++, j--) if (s[i] != s[j]) return false;
   return true;
}
```

Попробуем преобразовать эту функцию в рекурсивную. Для этого надо сформулировать рекурсивное определение по типу определения факториала. Очевидно, что строка из одного символа является палиндромом. Если символов больше, то строка является палиндромом, если выполняются два условия:

- первый и последний символы строки `s` совпадают;
- строка `s` без первого и последнего символа является палиндромом.

Для полноты картины и пустую строку будем считать палиндромом. Тогда мы по этому определению можем написать такую рекурсивную схему:

```
bool Palindrom(string s)
{  if (s пустая или имеет длина = 1) return true
   else if (первый символ s == последний символ s)
   return Palindrom(s без первого и последнего символа)
   else return false;
}
```

Теперь рассмотрим рекурсивную функцию, текст которой приведен в листинге 7.20.

Листинг 7.20. Рекурсивная функция `Palindrom`

```
bool Palindrom(string s)
{  int end = s.length()-1;
   if (s.length()=0 || s.length()=1) return true;
   else if (s[0] == s[end]) return Palindrom(s.substr(1, end-1));
   else return false;
}
```

Вызов функции может осуществляться, например, так:

```
cout << Palindrom("потоп") << endl;
```

или так:

```
string s = "потоп";
cout << Palindrom(s) << endl;
```

Теперь разберемся с эффективностью рекурсивной функции. Итеративная форма, очевидно, не делает никаких лишних действий: цикл, в котором сравниваются символы, выполняется либо до середины строки, либо сразу прекращается, если символы не совпали. Возврат выполняется единственный раз. Совсем другое дело — рекурсивная функция. Во-первых, всегда выполняется столько вызовов, сколько символов совпадает в начале и в конце строки. Столько же выполняется и возвратов. Во-вторых, при каждом вызове в стек помещаются все параметры и локальные переменные. В нашем случае это строка `s`. Рассмотрим этот процесс более подробно на примере приведенного выше вызова.

При первом вызове в стек помещается строка "потоп". Поскольку длина строки больше `0`, осуществляется проверка `s[0] == s[end]` (буква "п"). Результат равен `true`, поэтому выполняется оператор `return` с рекурсивным вызовом:

```
Palindrom(s.substr(1,end-1))
```

При втором вызове в стек помещается строка `s` без первого и последнего символа и локальная переменная `end`. Снова `s[0] == s[end]` (буква "о"), поэтому выполняется третий рекурсивный вызов со строкой из одного символа "т". Теперь происходит возврат `true` из первого условия `if`. Однако это выход только из третьего рекурсивного вызова, поэтому мы попадаем во второй вызов и тут же возвращаемся в первый вызов. Только после третьего возврата выполняется вывод полученного значения на экран (`true`, которое выводится как `1`).

Теперь рассмотрим рекурсивную реализацию двоичного поиска заданного числа в отсортированном по возрастанию массиве. Двоичный поиск по определению рекурсивен — его пошаговая схема выглядит так:

1. Вычисляем середину массива.
2. Если искомый элемент равен "среднему", то возвращаем результат.
3. Если искомый элемент меньше "среднего", то выполняем те же действия с левой половиной.
4. Если искомый элемент больше "среднего", то выполняем те же действия с правой половиной.

Шаги 3 и 4 и будут представлять рекурсивный вызов в программе. Переведем почти "дословно" рекурсивное определение на язык C++ и получим следующее определение функции (ее текст приведен в листинге 7.21).

Листинг 7.21. Рекурсивная функция двоичного поиска в массиве

```
int BinarySearch(const int a[], int b, int e, int k)
{  if (b == e) return -1;     //  не нашли
   int c = (b + e) / 2;       //  середина
   if (a[c] == k)  return c;
   else if (a[c] > k) c = BinarySearch(a, b, c, k);   //  левая половина
   else if (a[c] < k) c = BinarySearch(a, c+1, e, k); //  правая половина
}
```

Если элемент не найден, то функция возвращает `1`. В случае успешного поиска возвращается индекс найденного элемента. В рекурсивных вызовах параметры изменяются, каждый раз сужая область поиска ровно вдвое. Функция имеет первую форму и выполняет все действия на рекурсивном спуске. Пусть в вызывающей программе объявлен массив:

```
int d[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13 };
```

Тогда вызов

```
int k = BinarySearch(d, 0, 12, 7);
```

присвоит переменной `k` индекс числа `7`, равный `6`. Функция довольно эффективна, поскольку для массива длиной `n` элементов выполняет всего

$\log_2(n)$ вызовов функций, при каждом из которых в стек попадают всего 4 параметра.

Данная программа демонстрирует один из важнейших принципов, применяемых при разработке рекурсивных функций — "разделяй и властвуй": в теле функции выполняется два рекурсивных вызова, каждый из которых работает примерно с половиной данных. Подобную схему имеет и быстрая сортировка Хоара. Применяя этот же принцип, попробуем написать функцию вычисления минимума в массиве целых чисел. Итеративное определение не представляет сложностей. При этом даже не требуется оформлять поиск в виде функции — достаточно просто написать в программе следующий цикл:

```
for (int m = a[0], i = 1; i < n; i++) m =((a[i] < m) ? a[i]:m);
```

Данный цикл достаточно эффективен, поскольку за один проход массива гарантированно вычисляется минимум. Количество сравнений на 1 меньше количества элементов массива.

Рекурсивное определение несколько сложнее — без определения функции уже не обойтись, поскольку требуется иметь рекурсивный вызов, заменяющий цикл. Параметрами такой функции, как и в функции двоичного поиска, будет массив и два индекса, определяющие начало и конец обрабатываемого сегмента массива. Взяв за образец функцию двоичного поиска, напишем аналогичную функцию поиска минимума в массиве (ее текст приведен в листинге 7.22).

Листинг 7.22. Рекурсивная функция вычисления минимума в массиве

```
 int min(const int a[], int left, int right)
{  if (left == right) return a[left];    //  можно и a[right]
   int m = (left + right) / 2;     //  середина
   int x = min(a, left, m);        //  обработка левой половины массива
   int y = min(a, m+1, right);     //  обработка правой половины массива
   if (x < y) return x; else return y;
}
```

Данная функция имеет третью форму, выполняя действия как на рекурсивном спуске (до рекурсивного вызова), так и на рекурсивном возврате (после рекурсивного вызова). Внешне функция `min` очень похожа на функцию двоичного поиска, поэтому может создаться впечатление, что она, аналогично двоичному поиску, достаточно эффективно ищет требуемый нам минимум. На самом деле эта функция очень неэффективна. Мы можем посчитать количество и вложенность вызовов, объявив в функции статическую переменную. Увеличивая ее при каждом вызове и уменьшая при каждом возврате, можно получить достаточно полную картину рекурсивных вызовов и воз-

вратов. Оформим это в виде процедуры, выводящей на экран количество точек, равное уровню вложенности рекурсивного вывода:

```
void pp(int& tt, int l, int r)
{  cout << endl;
   for(int i = 0; i < tt; i++) cout << ".";
   cout << "min(" << l << ',' << r << ')';
}
```

Тогда полный "отладочный" вариант функции `min` теперь выглядит так:

```
int min(int a[], int left, int right)
{  static int tt = 0;                    //  счетчик уровня вложенности
   pp(++tt, left, right);                //  вложенность увеличивается
   if (left == right) return a[left];    //  можно и a[right]
   int m = (left + right) / 2;           //  середина
   int x = min(a, left, m);              //  обработка левой части
   pp(--tt, left, right);                //  вложенность уменьшается
   int y = min(a, m+1, right);           //  обработка правой части
   pp(--tt, left, right);                //  вложенность уменьшается
   if (x < y) return x; else return y;
}
```

Как показывает "прогон" программы, для массива из 8 элементов вызов `min(a, 0, 7)` на экран выводит следующее:

```
. min(0, 7)
. . min(0, 3)
. . . min(0, 1)
. . . . min(0, 0)
. . . min(0, 1)
. . . . min(1, 1)
. . . min(0, 1)
. . min(0, 3)
. . . min(2, 3)
. . . . min(2, 2)
. . . min(2, 3)
. . . . min(3, 3)
. . . min(2, 3)
. . min(0, 3)
. min(0, 7)
. . min(4, 7)
. . . min(4, 5)
. . . . min(4, 4)
. . . min(4, 5)
. . . . min(5, 5)
. . . min(4, 5)
```

```
. . min(4, 7)
. . . min(6, 7)
. . . . min(6, 6)
. . . min(6, 7)
. . . . min(7, 7)
. . . min(6, 7)
. . min(4, 7)
. min(0, 7)    //  завершение обработки всего массива
```

В этом фрагменте увеличение количества точек в очередной строке по сравнению с предыдущей означает рекурсивный вложенный вызов, уменьшение — возврат из вызова.

Попробуем написать более эффективную функцию. Надо вместо двух рекурсивных вызовов постараться оставить только один. Принцип "разделяй и властвуй" в своем "предельном" варианте диктует разделение исходного массива на две неравные части: единственный элемент (первый элемент массива) и остальной массив, размер которого на 1 меньше исходного. Текст более эффективной функции приведен в листинге 7.23.

Листинг 7.23. Эффективная рекурсивная функция поиска минимума

```
int min1(int a[], int begin, int end)
{  if (begin == end) return a[begin];
   int x = min1(a, begin+1, end);
   if (x < a[begin]) return x; else return a[begin];
}
```

Эта функция так же, как и аналогичная (см. листинг 7.22), выполняет действия как на рекурсивном спуске, так и на рекурсивном возврате. Она значительно эффективнее, поскольку выполняет ровно столько сравнений, сколько у нас имеется элементов в массиве `a`. В этом можно убедиться, выполнив "отладочную" версию:

```
. min(0, 9)
. . min(1, 9)
. . . min(2, 9)
. . . . min(3, 9)
. . . . . min(4, 9)
. . . . . . min(5, 9)
. . . . . . . min(6, 9)
. . . . . . . . min(7, 9)
. . . . . . . . . min(8, 9)
. . . . . . . . . . min(9, 9)    //  x = a[9]
. . . . . . . . . min(8, 9)
. . . . . . . . min(7, 9)
```

```
. . . . . . . min(6, 9)
. . . . . . min(5, 9)
. . . . . min(4, 9)
. . . . min(3, 9)
. . . min(2, 9)
. . min(1, 9)
. min(0, 9)
```

Как мы видим, количество вложенных рекурсивных вызовов равно количеству элементов массива. Глубина рекурсии тоже равна количеству элементов. Сначала до последнего элемента выполняется только рекурсивный спуск. Когда мы достигли последнего элемента — выполняется первый рекурсивный возврат и переменная `x` впервые получает значение. На первом шаге рекурсивного подъема переменная `x` сравнивается сама с собой! Выполняется оператор:

```
else return a[begin];
```

На рекурсивном подъеме при каждом возврате переменная `x` сравнивается с предыдущим элементом массива. Таким образом, как это обычно происходит в программировании, за увеличение скорости приходится расплачиваться расходом памяти (в данном случае — увеличением стека).

Рекурсия при обработке динамических структур данных

Динамические структуры данных, вообще говоря, делятся на три больших класса: *списки*, *деревья* и *графы*. Поскольку в каждом классе очень много разных видов (особенно это касается деревьев), мы рассмотрим только *линейные списки* и *двоичные деревья*. Эти структуры по сути своей являются рекурсивными структурами, писать для них рекурсивные функции достаточно легко.

Односвязный линейный список

Начнем с линейных односвязных списков, рассмотренных нами в *гл. 4*. Пусть информационная часть — поле целого типа:

```
struct Item { int info; Item* next; };
```

Для удобства дальнейших записей используем оператор:

```
typedef Item* link;
```

Как обычно, объявим заголовок списка:

```
link Head;    //  звездочка уже не нужна
```

Имея такие определения, напишем простую функцию вычисления количества элементов списка (ее текст приведен в листинге 7.24). Параметром является указатель-заголовок списка.

Листинг 7.24. Рекурсивный подсчет количества элементов списка

```
long count(link p)
{  if (p == 0) return 0;
   return 1 + count(p -> next);
}
```

Как мы видим, условием окончания является "пустой" указатель последнего элемента списка. Налицо и изменение параметра — переход по указателю на следующий элемент.

А теперь напишем функцию обхода всех элементов списка в прямом порядке. Для каждого элемента будем вызывать функцию обработки, указатель на которую, естественно, будем передавать в качестве параметра. Текст функции приведен в листинге 7.25.

Листинг 7.25. Функция обработки списка в прямом порядке

```
void visitor(link p, void f(link p))
{  if (p == 0) return 0;
   f(p);                    //  любая обработка
   visitor(p -> next, f);   //  рекурсивный переход к следующему
}
```

В качестве функции обработки может использоваться, например, функция вывода на экран. Или функция, проверяющая некоторые условия, на основании которых может формироваться новый список — вариантов много.

Рекурсия позволяет обходить односвязный список и в обратном порядке — и для этого не нужно вводить в структуру обратный указатель! Вспомним уже определенную функцию `ReadFile`, которая выводила строки файла в обратном порядке, и поступим аналогичным способом — поменяем порядок рекурсивного вызова и вызова функции обработки. Текст функции для обработки списка в обратном порядке приведен в листинге 7.26.

Листинг 7.26. Функция обработки списка в обратном порядке

```
void visitor(link p, void f(link p))
{  if (p == NULL) return 0;
   visitor(p -> next, f);   //  рекурсивный переход к следующему
   f(p);                    //  любая обработка
}
```

Обработка выполняется на рекурсивном возврате, а следовательно — в обратном порядке. Но понятно, что "бесплатного сыра не бывает" — мы расплачиваемся за простоту реализации использованием памяти под стек. Однако попробуйте написать аналогичную итеративную функцию — вы вынуждены будете явно использовать дополнительную память либо под стек, либо для обратных указателей. Тем самым наживете на свою голову лишние проблемы при отладке. А рекурсия "прячет" стек и позволяет легко и просто совместить прямой и обратный проходы. Текст функции для обработки списка в прямом и обратном порядке приведен в листинге 7.27.

Листинг 7.27. Функция прямой и обратной обработки списка

```
void visitor(link p, void pred(link p), void post(link p))
{  if (p == NULL) return 0;
   pred(p);
   visitor(p -> next, pred, post);   //  рекурсивный переход к следующему
   post(p);
}
```

Двоичное дерево

Двоичные деревья, так же как и списки, являются по определению рекурсивными структурами. Поэтому использовать рекурсию при обработке деревьев "сам Бог велел". Пусть узел дерева представлен следующей структурой:

```
struct Node {
   string name;     //  ключ
   Node *Left;      //  "младший левый сын"
   Node *Right;     //  "старший правый сын"
            };
```

Как обычно, левый узел содержит значение, меньшее корневого значения, а в правом — большее. Одной из первых задач, которые приходится решать при работе с деревьями, является вывод значений в возрастающем порядке. В "классической" литературе [9, 14, 16, 39, 41, 45] такие задачи называются "обходом" дерева. Поскольку у нас в каждом узле фигурируют три сущности: корень `Root`, "младший сын" `Left`, "старший сын" `Right`, — мы имеем три разных способа обхода дерева.

1. `Root`, `Left`, `Right`. У Вирта [9] такой порядок называется `preOrder`.
2. `Left`, `Root`, `Right`. У Вирта такой порядок называется `inOrder`.
3. `Left`, `Right`, `Root`. У Вирта такой порядок называется `postOrder`.

Три вида обхода в точности соответствуют трем формам рекурсивных функций. Естественно использовать для рекурсии условие продолжения. Параметром является указатель на узел — при первом вызове это будет указатель на корень всего дерева. Текст функций обхода деревьев приведен в листинге 7.28.

Листинг 7.28. Функции обхода дерева

```
typedef Node* link;
void preOrder(link p, void visit(link p))
{  if (p != NULL)              //  условие продолжения
   {  visit(p);                //  обработка узла
      preOrder(p -> Left);     //  пошли налево
      preOrder(p -> Right);    //  пошли направо
   }
}
void inOrder(link p, void visit(link p))
{  if (p != NULL)              //  условие продолжения
   {  inOrder(p -> Left);      //  пошли налево
      visit(p);                //  обработка узла
      inOrder(p -> Right);     //  пошли направо
   }
}
void postOrder(link p, void visit(link p))
{  if (p != NULL)               //  условие продолжения
   {  postOrder(p -> Left);     //  пошли налево
      postOrder(p -> Right);    //  пошли направо
      visit(p);                 //  обработка узла
  }
}
```

Если вместо функции `visit` проставить оператор вывода значения, то функция `inOrder` как раз выполнит нашу задачу — выведет все значения в возрастающем порядке.

Обратите внимание на то, что во всех функциях обхода используется условие продолжения — рекурсивные вызовы повторяются до тех пор, пока указатель указывает на узел. Такая схема является достаточно типичной при обработке двоичных деревьев. Попытки реализовать аналогичные процедуры итеративного вида убеждают нас, что без явного использования стека обойтись трудно.

Нужно сказать, что фактически все процедуры обработки деревьев в той или иной мере используют один из вариантов обхода — как при вставке, так и при удалении необходимо выполнять поиск места в дереве, где вы-

полнять операцию. В качестве примера приведем функцию поиска в бинарном дереве заданной строки [16]. Функция поиска имеет два параметра: указатель на узел дерева и искомую строку. Возвращает указатель на узел, в котором найдена заданная строка. Если строка не найдена, то возвращается NULL. В этой функции естественно используется первая форма рекурсивной процедуры: сначала выполняется действие (сравнение заданной и искомой строки), а затем происходит рекурсивный вызов. Текст функции приведен в листинге 7.29.

Листинг 7.29. Функция поиска в двоичном дереве

```
link Search(link p, char *name)
{  int cmp;
   if (p == NULL) return NULL;     //  не нашли
   cmp = strcmp(name, p -> name);
   if (cmp == 0) return p;         //  нашли
   else if (cmp < 0)
   Search(p -> left, name);        //  пошли налево
   else
   Search(p -> right, name);       //  пошли направо
}
```

Приведем еще несколько простых рекурсивных алгоритмов, применяемых к двоичным деревьям. Например, подсчет узлов в дереве можно осуществить выполнив функцию, текст которой приведен в листинге 7.30.

Листинг 7.30. Функция подсчета узлов в дереве

```
unsigned long count(link p)
{  if (p == NULL) return 0;
   return count(p -> Left) + 1 + count(p -> Right);
}
```

Возвращаемый тип unsigned long позволяет нам обрабатывать очень большие деревья. Такой же простой является функция подсчета высоты дерева (ее текст приведен в листинге 7.31).

Листинг 7.31. Функция вычисления высоты дерева

```
typedef unsigned long ul;
ul h(link p)
{  if (p == 0) return 0;      //  пустое дерево или дошли до листа
   ul x = h(p -> Left);       //  высота левого поддерева
   ul y = h(p -> Right);      //  высота правого поддерева
```

```
   if (x > y) return x+1;     //  +1 от корня к левому поддереву
   else return y+1;           //  +1 от корня к правому поддереву
}
```

И в той и в другой функции используется условие окончания — рекурсивные вызовы прекращаются.

Параметры в рекурсивных функциях

Параметры в рекурсивные функции можно передавать во всех возможных формах. В рассмотренных ранее примерах они передавались по значению. Указатели обычно используются при обработке массивов, однако сами-то указатели передаются по значению. Возникают следующие вопросы:

- Можно ли присвоить начальные значения параметрам рекурсивных функций?
- К каким последствиям может привести передача в рекурсивную функцию параметра по ссылке?

Ответ на первый вопрос очевидно положительный — простой пример с факториалом подтверждает это. Допустим, в нашей программе надо вычислять факториалы, и чаще всего требуется значение 10!. Тогда мы можем явно прописать `10` как значение по умолчанию:

```
long Fact(int k = 10)
{ if (k > 0) return k * Fact(k-1); else return 1; }
```

Вызов функции осуществляется самым обычным способом:

```
cout << Fact(6) << endl;
cout << Fact() << endl;
```

Первый вызов выводит на экран число `720`, а второй — значение 10!, равное `3628800`. Как обычно, начальное значение может вычисляться с помощью некоторой функции. И наиболее интересный случай — вызов той же самой рекурсивной функции для вычисления начального значения. Только, как мы помним, надо впереди прописать прототип, например:

```
double Fact(double k);
double Fact(double k = Fact(4))
{ if (k > 0) return k * Fact(k-1); else return 1; }
```

Обратим внимание на следующие особенности:

- вызов функции в списке параметров не является рекурсивным, несмотря на то, что сама функция является рекурсивной. Фактически вызов той же функции на месте параметра означает F(F(x)). В нашем случае имеем (4!)! = 24!;

- мы поменяли тип возвращаемого значения на `double`, поскольку 24! — значительно больше, чем предельно допустимая величина `unsigned long`. Кстати, знаете ли вы, какой максимальный факториал вычисляется на Pentium разрядностью 32? Всего 1754!. Это число имеет порядок 10^{4930}, а тип его должен быть `long double`;
- нам потребовалось поменять и тип параметра на `double`, чтобы присвоение значения параметру произошло корректно. Иначе возникают проблемы при выполнении.

Заметим, что и в рекурсивных функциях можно использовать фокус с операцией "запятая" при вызове (см. листинг 2.5) — на месте единственного параметра может в скобках через запятую стоять несколько выражений. В функцию, как обычно, передается последнее выражение.

Для ответа на второй вопрос, определим функцию вычисления факториала с параметром-ссылкой (ее текст приведен в листинге 7.32).

Листинг 7.32. Функция передачи параметра-ссылки в рекурсивную функцию

```
double Fact(unsigned int& k)
{  if (k == 0) return 1;
   return (k * Fact(k-1));
}
```

Если у нас определена переменная

```
int n = 5;
```

то вызов `Fact(n)` выдаст значение `120`, и `n` останется равным `5`. Тогда в чем проблема? Немного изменим функцию:

```
double Fact(unsigned int& k)
{  if (k == 0) return 1;
   return (k * Fact(--k));    //  рекурсивный вызов
}
```

Рекурсивный вызов `Fact(k-1)` превратился в `Fact(--k)`. С точки зрения вычисления факториала ничего не изменилось. Однако после этого вызова значение переменной `n` уменьшится на 1. Функция имеет побочный эффект — изменение переменной, переданной в качестве параметра. Поэтому после вызова `Fact(n)` переменная `n` станет равной нулю. Вариант функции с передачей параметра по значению работает совершенно корректно и при втором случае рекурсивного вызова.

ЧАСТЬ II

Объектно-ориентированное программирование

Глава 8. Создание простых типов

Глава 9. Динамические классы

Глава 10. Исключения — что это такое

Глава 11. Наследование

Глава 12. Обобщенное программирование

ГЛАВА 8

Создание простых типов

Подпрограммы с самого начала служили средством расширения языка программирования. Если в языке не хватало какой-то функциональности, создавалась библиотека подпрограмм, эту функциональность обеспечивающая. Это очень хорошо видно на примере Фортрана, для которого было создано огромное количество библиотек, особенно по численным методам. Однако постепенно программистское сообщество "сообразило", что расширять язык можно не только и не столько подпрограммами, сколько новыми типами данных.

C++ разрешает программисту определить свои собственные типы данных, которые практически не отличаются от встроенных. Однако, прежде чем перейти непосредственно к изучению конструкций языка C++ для определения новых типов, давайте ответим на вопрос, что такое "тип данных". Во-первых, очевидно, что в это понятие входит множество объектов-значений. Например, целый тип включает множество целых чисел. Во-вторых, сами по себе значения никому не нужны, нужна возможность оперировать этими значениями. Мы имеем в языке большой набор операций с целыми числами. Принцип полиморфизма требует, чтобы подобный набор операций был определен при конструировании нового типа. В-третьих, объект конструируемого типа может иметь сложную структуру и состоять из более простых, встроенных или определенных ранее, типов данных.

Перегрузка операций

Мы уже знакомы с некоторыми конструкциями языка C++, с помощью которых определяется новый тип данных: `enum` и `struct`. Более того, знаем, что можно в качестве элемента структуры задать функцию, связав ее тем самым со структурой. Однако при определении нового типа данных было бы удобно определить не функции, а операции.

Согласитесь, что сложение, например, комплексных чисел более естественно записать так:

```
a = b + c;
```

а не так:

```
a = add(b, c);
```

Поэтому нам необходима перегрузка операций. Язык C++ позволяет это сделать. Собственно, мы уже пользовались перегруженными операциями — это операции ввода/вывода << и >>, которые "по совместительству" являются операциями сдвига. Вернее, наоборот, в языке определены операции сдвига, а в библиотеке iostream эти операции перегружены.

C++ во многом ограничивает перегрузку операций. Первое и самое сильное ограничение заключается в том, что нельзя писать новые функции-операции. Допускается перегружать только встроенные в язык операции. Например, нельзя ввести новый значок (например, ** — как в Фортране) для операции возведения в степень, но можно использовать один из тех, которые определены в языке (например, ^). Кроме того, запрещено перегружать следующие операции:

- . — (точка) селектор компонента объекта;
- .* — обращение к компоненту через указатель;
- ?: — условная операция;
- :: — операция указания области видимости;
- sizeof — операция вычисления размера типа или выражения;
- # — (решетка) операция препроцессора;
- ## — операция сцепления препроцессора.

Кроме того, перегрузка операций допускается только для нового типа данных — нельзя перегрузить операцию для встроенного типа. Например, нельзя перегрузить операцию ^ как операцию возведения в степень ни для double, ни для какого другого числового типа. Прототип такой функции-операции мог бы быть таким:

```
double operator^(double a, int b)
```

Однако в системе Visual C++ 6 транслятор выдает сообщение об ошибке C2803:

```
'operator ^' must have at least one formal parameter of class type
'operator ^' должен иметь, по крайней мере, один параметр типа класса
```

Таким образом, чтобы ввести операцию возведения в степень, нам придется сначала объявить новый тип данных либо объявив структуру, либо с помощью конструкции enum.

Операции присваивания =, индексирования [], вызова функции () и доступа по указателю -> допускается перегружать только методами класса. Однако и при отсутствии перегрузки эти операции практически всегда корректно работают с новым типом. Остальные операции допускается перегружать как методами класса, так и внешними функциями. Перегрузки внешними функциями и "внутренними" довольно существенно отличаются.

Прототип функции-операции выглядит следующим образом:

```
тип operator@(список параметров);
```

где @ — символ операции. Синтаксис и приоритет перегружаемой операции также изменить нельзя. Это означает, что унарную операцию нельзя сделать при перегрузке бинарной и наоборот. При перегрузке операций запрещается использовать параметры по умолчанию, и нет никакой возможности перегрузить операцию, задав список параметров переменной длины — перегружаемые операции либо унарные, либо бинарные. Прототип бинарной операции, перегружаемой внешней функцией, выглядит так:

```
тип operator@(параметр_1, параметр_2);
```

Естественно, параметры можно передавать любым удобным способом. Обращение к определенной таким образом операции выполняется двумя различными способами:

- инфиксная форма `параметр_1@параметр_2`;
- функциональная форма `operator@(параметр_1, параметр_2)`.

Прототип унарной операции имеет вид:

```
тип operator@(параметр);
```

а обращение к перегруженной операции выглядит так:

- инфиксная форма `@параметр`;
- функциональная форма `operator@(параметр)`.

Без использования классов мы можем перегружать операции только для конструкций `enum` и `struct`. Рассмотрим сначала первый вариант.

Примечание

В системе Borland C++ 3.1 не реализована возможность перегрузки операций для перечислимого типа. Поэтому все примеры проверялись в системах Visual C++ 6, Borland C++ Builder 6 и Borland C++ 5.

Перегрузка операций для *enum*

Пусть у нас определен следующий перечислимый тип:

```
enum Decade
{ zero, one, two, three, four, five, six, seven, eight, nine };
```

В данном случае мы просто поименовали целые числа от нуля до девяти. Предположим, нам требуется реализовать арифметику с этими числами по модулю 10. Желательно при перегрузке операций сохранять традиционную семантику. Таким образом, функция сложения должна работать в соответствии с табл. 8.1, содержащей результаты сложения данных чисел по модулю 10.

Таблица 8.1. *Сложение по модулю 10*

+	zero	one	two	three	four	five	six	seven	eight	nine
zero	zero	one	two	three	four	five	six	seven	eight	nine
one	one	two	three	four	five	six	seven	eight	nine	zero
two	two	three	four	five	six	seven	eight	nine	zero	one
three	three	four	five	six	seven	eight	nine	zero	one	two
four	four	five	six	seven	eight	nine	zero	one	two	three
five	five	six	seven	eight	nine	zero	one	two	three	four
six	six	seven	eight	nine	zero	one	two	three	four	five
seven	seven	eight	nine	zero	one	two	three	four	five	six
eight	eight	nine	zero	one	two	three	four	five	six	seven
nine	nine	zero	one	two	three	four	five	six	seven	eight

Совершенно очевидно, что заголовок функции-операции сложения при передаче параметров по ссылке должен выглядеть так:

```
Decade operator+(const Decade &a, const Decade &b)
```

Указывая константные параметры, мы подчеркиваем тот факт, что функция не изменяет свои аргументы в теле. Однако попытка реализовать функцию "в лоб"

```
Decade operator+(const Decade &a, const Decade &b)
{ return Decade((a + b) % 10); }
```

не проходит — в системе Visual C++ 6 вызов функции завершается аварийно по причине переполнения стека. Дело в том, что знак + внутри функции воспринимается отнюдь не как сложение целых чисел, а как рекурсивный вызов определяемой функции-операции. Поэтому требуется "обмануть" компилятор с помощью преобразования типа (листинг 8.1).

Листинг 8.1. Перегрузка сложения, версия 1

```
Decade operator+(const Decade &a, const Decade &b)
{ Decade t = Decade((int)a+(int)b); return Decade(c % 10); }
```

Переменная `t` определена исключительно для упрощения вычисления возвращаемого выражения. Для того чтобы компилятор не воспринимал операцию `+` как рекурсивный вызов, нам потребовалось явно указать преобразование в `int` для обоих параметров, а потом прописать обратное преобразование в `Decade`. Преобразование в операторе `return` также обязательно, поскольку без него Visual C++ 6 выдает сообщение об ошибке:

```
error C2440:'return': cannot convert from 'int' to 'enum Decade'
'return': не могу конвертировать из 'int' в 'enum Decade'
```

Замечание

Понятно, что на самом деле это не является реальным преобразованием, однако по стандарту C++ мы обязаны написать формальное преобразование.

Вызов данной функции

```
Decade A, B;
A = seven;
B = six;
cout << A + B << endl;
```

выводит на экран значение (7 + 6) % 10, равное `3`. Обратите внимание, что мы не перегружали ни операцию присваивания, ни операцию вывода `<<`, однако обе они корректно срабатывают. Более того, для перечислимого типа правильно работает и инициализация:

```
Decade A = seven, B = six;
cout << operator+(A, B) << endl;
```

Это и понятно — перечислимый тип является поименованным подмножеством целого типа. В данном случае мы продемонстрировали второй вариант вызова — функциональный.

Опытные программисты часто поступают следующим образом: перегружают операцию `+=`, а затем с помощью нее перегружают "чистую" операцию сложения. Сделаем и мы то же самое. Очевидно, что функция должна иметь два аргумента: первый аргумент изменяется и становится равным сумме аргументов, второй аргумент — не изменяется, а операция выдает значение, равное сумме аргументов. Поэтому первый аргумент передается по ссылке, а второй — по значению (или по ссылке-константе). Неясно, что возвращать, поэтому реализуем пока возврат значения типа `Decade`. Текст функции приведен в листинге 8.2.

Листинг 8.2. Перегрузка сложения с присваиванием (неправильная)

```
Decade operator+=(Decade &a, const Decade &b)
{ return Decade(a = Decade((a + b) % 10)); }
```

Мы снова "обманули" компилятор: использовали операцию сложения + для реализации операции сложения с присваиванием. Проверим работу нашей операции:

```
cout << (A += B) << endl;
```

Мы написали выражение в скобках, поскольку приоритет операции += меньше приоритета операции <<, поэтому без скобок возникают ошибки трансляции.

Так как встроенная операция += является многократной, то проверим многократность реализованной операции. Оказывается, не все так просто:

```
decade A = one, B = one, D = one;
cout << (A += B += D) << endl;       //  B = two, A = three
cout << ((A += B) += D) << endl;     //  не транслируется
```

Встроенная операция += правильно работает с целыми числами в обоих случаях, а наша во втором варианте выдает ошибку трансляции! Разобраться в проблеме помогает функциональный вызов. Первый вариант эквивалентен:

```
cout << operator+=(A, operator+=(B, D)) << endl;
```

Вычисляется "внутренний" вызов `operator+=(B, D)`, значение `B` становится равным `two` и используется как аргумент "внешнего" вызова, изменяя `A`.

Второй вариант эквивалентен:

```
cout << operator+=(operator+=(A, B), D) << endl;
```

Так как первый параметр передается по ссылке, становится понятно, что операция должна возвращать ссылку, а не значение. Правильный вариант нашей операции приведен в листинге 8.3.

Листинг 8.3. Перегрузка сложения с присваиванием

```
Decade& operator+=(Decade &a, const Decade &b)
{ return Decade(a = Decade((a + b) % 10)); }
```

Реализация сложения посредством операции += теперь совсем проста и приведена в листинге 8.4.

Листинг 8.4. Перегрузка сложения, версия 2

```
Decade operator+(const Decade &a, const Decade &b)
{ Decade t = a; return Decade(t += b); }
```

Как видим, читабельность функции здорово улучшилась. Теперь мы можем перегрузить операцию сложения `Decade` с типом `int` (листинг 8.5).

Листинг 8.5. Перегрузка сложения `Decade` с типом `int`

```
Decade operator+(const Decade &a, const int &b)
{ Decade t = a; return Decade(t += Decade(b % 10)); }
Decade operator+(const int &a, const Decade &b)
{ Decade t = Decade(a % 10); return Decade(t += b); }
```

Примечание

Интересно, что при попытке передавать параметры по значению в системе Visual C++ 6 инфиксная форма вызова с первым аргументом типа `Decade`, а вторым `int` не транслируется! Visual C++ 6 разрешает для этого варианта использовать только функциональную форму вызова, например, `operator(A, 1)`. Это, конечно, минус Visual C++ 6, т. к. и системы фирмы Borland, и Visual C++ 7 все транслируют и выполняют, не выдавая никаких сообщений об ошибках. Таким образом, в каждой системе встречаются свои собственные недостатки, которые программист вынужден "обходить" тем или иным способом.

Совершенно аналогично реализуется перегрузка других бинарных операций, поэтому мы не будем на этом останавливаться. Попробуем реализовать операцию инкремента `++`. Как известно *(см. гл. 1)*, префиксная и постфиксная формы обладают разной семантикой. Префиксная форма реализуется совершенно элементарно (текст функции приведен в листинге 8.6).

Листинг 8.6. Перегрузка префиксного инкремента

```
Decade operator++(Decade &a)
{ return Decade((a += one) % 10); }
```

Семантика постфиксной формы реализуется следующей функцией:

```
Decade operator++(Decade &a)
{  Decade t = a;                    //  запомнили
   a = Decade((a += one) % 10);     //  изменили
   return Decade(t);                //  вернули первоначальное
}
```

Сначала сохраняется текущее значение аргумента, потом аргумент увеличивается, затем возвращается сохраненное ранее значение. Все правильно, однако программа не транслируется, поскольку мы имеем две реализации для одного прототипа. Для того чтобы эти две формы операции инкремента различались, в прототип постфиксной формы надо добавить фиктивный целый аргумент, который при вызове операции никогда не используется. Текст функции-операции, с добавлением фиктивного аргумента, приведен в листинге 8.7.

Листинг 8.7. Перегрузка постфиксного инкремента

```
Decade operator++(Decade &a, int)
{  Decade t = a;
   a = Decade((a += one) % 10);
   return Decade(t);
}
```

Следующая программа демонстрирует работу обеих форм операции инкремента:

```
int main(void)
{  Decade A = seven, B = six;
   cout << ++A << endl;    //  выводит eight
   cout << A++ << endl;    //  выводит eight
   cout << A << endl;      //  выводит nine
   return 0;
}
```

Таким образом, как можно видеть, даже для такой элементарной конструкции, как объявление перечислимого типа данных, C++ предоставляет достаточно развитые возможности по адаптации языка к новому типу.

Перегрузим операцию вывода <<, чтобы вывод выглядел в терминах нашего типа данных `Decade`. Операция должна получать два параметра — ссылку на поток вывода и выводимый аргумент, а возвращать — ссылку на поток. Текст функции-операции приведен в листинге 8.8.

Листинг 8.8. Вывод типа `Decade`

```
ostream& operator << (ostream &out, const Decade &a)
{  char *s[] = { "zero", "one", "two", "three", "four",
                 "five", "six", "seven", "eight", "nine"
               };
   out << s[a];
   return out;
}
```

При использовании этой операции на экран будут выводиться не цифры, а символьные обозначения констант нашего типа `Decade`.

Снова структуры

Вспомним, что структура по умолчанию является классом. Поэтому при использовании структур появляется возможность перегрузки операций для выполнения необходимых действий над структурами. Здесь мы уже вплот-

ную приближаемся к конструированию новых типов данных, которые Б. Страуструп [37] называет "конкретными типами".

Рассмотрим сначала перегрузку операций внешними функциями. Пусть в программе объявлена структура, представляющая беззнаковое рациональное число. Чтобы не задумываться в дальнейшем о величине числителя и знаменателя, используем тип `unsigned long`.

```
struct Rational { unsigned long num, denum; };
```

Наличие такого описания позволяет использовать слово `Rational` в качестве типа переменных, параметров и возвращаемого значения. Для правильной реализации операций над рациональными числами нам потребуется функция вычисления наибольшего общего делителя (НОД) и функция сокращения числителя и знаменателя. Текст обеих функций приведен в листинге 8.9.

Листинг 8.9. Функции "Наибольший общий делитель" и сокращения

```
typedef unsigned long ULONG;     //  чтобы меньше писать
struct Rational { ULONG num, denum; };
ULONG gcd(ULONG x, ULONG y)
{ if (y == 0) return x; return gcd(y, x%y); }
void reduce(Rational &c)
{  ULONG t = ((c.num > c.denum) ? gcd(c.num, c.denum)
                : gcd(c.denum, c.num));
   c.num /=t; c.denum /=t;       //  сокращаем
}
```

Напомним, что при первом вызове функции НОД должно выполняться условие `x > y`.

Пойдем уже по известному пути и сразу перегрузим операцию сложения с присваиванием +=. Как мы помним, сложение выполняется по следующему правилу:

```
(a/b) + (c/d) = (ad + cb)/(bd)
```

После выполнения операций надо сократить полученную дробь. Для этого мы и определили вспомогательные функции. Чтобы не отвлекаться на неважные в данном случае детали, мы не будем обрабатывать аварийные случаи. Текст операции сложения с присваиванием приведен в листинге 8.10.

Листинг 8.10. Операция сложения с присваиванием рациональных чисел

```
Rational& operator+=(Rational &a, const Rational &b)
{  a.num = a.num*b.denum + b.num*a.denum;    //  числитель
```

```
   a.denum = a.denum*b.denum;                  //  знаменатель
   reduce(a);                                  //  сокращаем
   return a;
}
```

Как и для перечислимого типа `Decade`, операция сложения с присваиванием должна иметь два параметра, причем первый параметр должен передаваться по ссылке. Возвращать такая функция должна тоже ссылку.

Перегрузка "чистой" операции сложения с учетом уже написанных функций и операции становится совсем простой (ее текст приведен в листинге 8.11).

Листинг 8.11. "Чистая" операция сложения рациональных чисел

```
Rational operator+(Rational a, Rational b)
{ Rational t = a; t += b; reduce(t); return t; }
```

Для полноты картины перегрузим еще операции ввода и вывода (листинг 8.12).

Листинг 8.12. Ввод/вывод рациональных чисел

```
// вывод рациональных чисел
ostream& operator << (ostream& t, Rational a)
{ return (t << '(' << a.num << '/' << a.denum << ')'); }
// ввод рациональных чисел
istream& operator >> (istream& t, Rational& a)
{ t >> a.num; t >> a.denum; reduce(a); return t; }
```

Так как мы разрешаем пользователю вводить любые возможные значения, то необходим вызов функции сокращения. Проверим наши определения функций:

```
void main(void)
{  Rational A, B, C;     //  объявление рациональных переменных
   A.num = 1;
   A.denum = 2;
   B.num = 1;
   B.denum = 3;
   C = A + B;            //  сложение рациональных чисел
   cout << C << endl;    //  вывод результата
}
```

На экран выводится `(7/6)`. Обратите внимание, что мы не перегружали операцию присваивания. Тем не менее, C++ разрешает использовать ее для присваивания переменных нового типа — структуры-то присваивать можно.

Аналогично, не перегружая операцию индексирования, мы можем объявить массив рациональных чисел и работать с индексированными переменными:

```
void main(void)
{  Rational d[5], C;
   cout << "Введите 5 рациональных чисел: num denum <enter>" << endl;
   for (int i = 0; i < 5; i++) cin >> d[i];          //  ввод массива
   for (i = 0; i < 5; i++) cout << d[i] << endl;     //  вывод массива
   C = d[0] + d[1] + d[2] + d[3] + d[4];             //  сумма элементов
   cout << C << endl;
}
```

При запуске программы введем пять рациональных чисел. В одной строке через пробел сначала указывается числитель, затем знаменатель. После знаменателя нажимается клавиша <Enter>:

```
1 1
2 2
3 3
4 4
5 5
```

Программа сначала выводит на экран в столбик значения введенных рациональных чисел, а затем вычисленную сумму элементов массива. Результат выглядит так:

```
(1/1)
(1/1)
(1/1)
(1/1)
(1/1)
(5/1)
```

Здесь мы наблюдаем работу функции сокращения, которая вызывается при вводе. Перегрузим теперь другие операции. Реализация операции умножения (ее текст приведен в листинге 8.13) несколько проще реализации сложения.

Листинг 8.13. Умножение рациональных чисел

```
Rational operator*(Rational a, Rational b)
{  Rational c;
   c.num = a.num*b.num;           //  вычисляем числитель
   c.denum = a.denum*b.denum;     //  вычисляем знаменатель
   reduce(c);                     //  сокращаем
   return c;                      //  возвращаем
}
```

Перегрузим теперь операцию умножения для работы с целыми числами `unsigned long` (листинг 8.14). Нам требуется две функции.

Листинг 8.14. Умножение операции `Rational` и типа `unsigned long`

```
Rational operator*(Rational a, const ULONG &b)
{  Rational c;
   c.num = a.num*b;
   c.denum = a.denum;
   reduce(c);     //  сокращаем
   return c;      //  возвращаем
}
Rational operator*(const ULONG &a, Rational b)
{  Rational c;
   c.num = a*b.num;
   c.denum = b.denum;
   reduce(c);     //  сокращаем
   return c;      //  возвращаем
}
```

При трансляции никаких сообщений не выдается — список параметров разный, и наблюдается "нормальная" перегрузка функций. Проверим выполнение:

```
int main(void)
{  Rational A, B, C;    //  объявление рациональных переменных
   cout << "Задайте число: числитель знаменатель <Enter>";
   cin >> A;
   cout << "Задайте число: числитель знаменатель <Enter>";
   cin >> B;
   cout << A * 3 << endl;
   cout << 5 * A << endl;
   ULONG d = 8;
   cout << B * d << endl;
   cout << d * A << endl;
   return 0;
}
```

При вводе `A = (1/2)` и `B = (1/4)` получаем на экране следующие результаты:

```
(3/2)
(5/2)
(2/1)
(4/1)
```

Аналогичным образом можно перегрузить и все остальные бинарные арифметические операции. Таким образом, мы получим возможность вычислять смешанные выражения целых и рациональных чисел.

Совершенно аналогично реализациям для перечислимого типа `Decade` можно реализовать и операции инкремента и декремента. Однако прежде нам придется перегрузить операцию `+=` для работы с целыми числами, как это сделано для операции умножения (листинг 8.15).

Листинг 8.15. Перегрузка операции сложения `Rational` с типом `int` с присваиванием

```
Rational& operator+=(Rational &a, const ULONG &b)
{ a.num = a.num + b; reduce(a); return a; }
```

Обратите внимание, что нам требуется только одна функция, реализующая вычисления в выражениях вида `a += 2`. Выражения вида `2 += a` не допускаются в C++, поэтому вторая функция не нужна. Теперь реализация перегруженных функций инкремента осуществляется совсем просто (листинг 8.16).

Листинг 8.16. Перегрузка инкремента

```
Rational operator++(Rational &a)
{ a += 1; reduce(a); return a; }
Rational operator++(Rational &a, int)
{ Rational t = a; a += 1; reduce(a); return t; }
```

Использовать определенные операции можно точно так же, как встроенные операции для числовых типов, например:

```
cout << ++A << endl;
cout << A++ << endl;
cout << A << endl;
```

Если переменная `A` имеет значение `(2/3)`, на экран будет выведено:

```
(5/3)
(5/3)
(8/3)
```

Покажем еще одну из операций сравнения, например, операцию `>` (больше). Ее определение приведено в листинге 8.17. Операция бинарная, параметров два, передаются по значению. Результат "истина", если первый аргумент больше второго. Для сравнения рациональных чисел их надо привести к общему знаменателю и сравнить числители.

Листинг 8.17. Операция сравнения >

```
bool operator>(Rational a, Rational b)
{ return (a.num*b.denum > b.num*a.denum); }
```

Остальные операции сравнения реализуются совершенно аналогично.

Конструкторы

В *гл. 3* у нас была определена структура `Person`. Наличие поля — символьного массива вызвало проблемы, поэтому заменим символьный массив полем типа `string`:

```
struct Person { string fio; Date BirthDay; };
```

Однако выясняется, что инициализировать такую структуру не удается.

```
Person a = {"Страуструп", { 1955, 12, 21 } };
```

Такая строка при трансляции в Visual C++ 6 сопровождается аж двумя сообщениями об ошибках. Но вспомним, что структура — это класс, и мы можем объявлять в структуре функции. В C++ разработан специальный механизм, позволяющий инициализировать объекты при объявлении — конструктор. *Конструктор* — это особая функция, имеющая имя, совпадающее с типом структуры. Количество и типы параметров конструктора могут быть любые, но обычно параметры используются для заполнения полей структуры. Однако конструктор имеет одно важнейшее свойство, существенно отличающее его от всех остальных функций: конструкторы не возвращают результата.

Напишем конструктор для структуры `Person`, позволяющий инициализировать поле `fio` символьной константой. Заголовок конструктора может быть таким:

```
Person(char s[])
```

или таким:

```
Person(char *s)
```

Реализация тоже тривиальна (ее текст приведен в листинге 8.18).

Листинг 8.18. Простейший конструктор в структуре

```
struct Person
{  string fio;
   Date BirthDay;
   Person(char *s) { fio = s; }
};
```

Как обычно, функция, элемент структуры, имеет неограниченный доступ ко всем полям. Конструктор позволяет инициализировать переменную типа `Person` так:

```
Person B ("Привет большой и горячий!");
```

Проходит и традиционная форма инициализации:

```
Person B = "Привет большой и горячий!";
```

На самом деле такая запись означает неявный вызов конструктора:

```
Person B = Person("Привет большой и горячий!");
```

При этом в правой части создается временный (анонимный) объект, который уничтожается после присвоения значения объекту `B`.

Таким образом, определение элементарного конструктора в структуре облегчает программисту жизнь — инициализация упрощается. Однако тут нас подстерегает "подводный камень": неожиданно оказывается, что объявления без инициализации

```
Person t;
```

являются ошибочными. Между тем, пока мы не объявляли конструктор, проблем с объявлениями не было.

Как известно, любую переменную-структуру мы можем объявить тремя способами:

- ❒ без инициализации;
- ❒ с инициализацией полей, всех или не всех;
- ❒ с инициализацией другой структурой.

Пока конструкторов нет, работают правила C++. Как только программист определит хотя бы один конструктор, C++ "умывает руки" и полагается на программиста. В такой ситуации нам необходимо объявлять три конструктора, которые конструируют объект данного типа в указанных ситуациях.

1. *Конструктор по умолчанию.* Это конструктор без параметров и, как правило, с пустым телом.
2. *Конструктор инициализации.* Это конструктор с любым необходимым количеством параметров произвольного типа. Для структуры `Person` это конструктор с одним параметром типа `char[]`.
3. *Конструктор копирования.* Это конструктор с одним параметром-ссылкой на объект данного типа. Инициализирует поля структуры данными из структуры-параметра.

При этом, естественно, используется перегрузка конструкторов. Напишем полное определение структуры, заодно прописав в конструкторе инициализации параметр для поля `BirthDay`. Текст структуры приведен в листинге 8.19.

Листинг 8.19. Структура `Person` с тремя конструкторами

```
struct Person { string fio; Date BirthDay;
   Person(){}                        //  поле по умолчанию
   Person(char *s) { fio = s; }      //  инициализация только одного поля
   Person(const char *s, const Date &d)
   { fio = s; BirthDay = d; }        //  инициализация всех полей
   Person(const Person &p)           //  копирование
   { fio = p.fio; BirthDay = p.BirthDay; }
};
```

Обратите особое внимание на то, что в качестве параметра в конструкторе копирования применяется ссылка на объект такого же типа. Использовать передачу по значению нельзя. Это и понятно — как же мы передадим в качестве параметра объект, который еще не сконструирован?! Поэтому определение такого конструктора с параметром по значению является серьезной ошибкой.

Вообще-то конструктор копирования всегда создается системой по умолчанию. Это означает, что если нам не требуется при копировании реализовать дополнительные действия, то сам копирующий конструктор можно не писать — объявления вида

```
Person t = s;
```

или

```
Person t(s);
```

прекрасно работают и без него. Только в том случае, когда нам, помимо простого копирования полей структуры требуется реализовать еще какие-нибудь действия (например, проверки на допустимость аргументов), мы должны определить конструктор копирования.

Определение выглядит замечательно, все транслируется без ошибок. Но тут оказывается, что мы не всегда можем написать инициализацию поля `BirthDay`.

```
Person A;         //  работает
Person B(A);      //  работает
Person D = B;     //  работает
Person F = { "Это фамилия", { 2003, feb_, 22 } };    //  не работает
```

Работает только вариант с передачей в качестве второго параметра уже объявленной переменной типа `Date`:

```
Date dd = { 2002, feb_, 22 };
Person F("Это фамилия", dd);
```

Можно предложить несколько решений этой проблемы. Например, объявить конструктор с четырьмя параметрами:

```
Person(const char *s, unsigned int y, Month m, unsigned int d)
{ fio = s; BirthDay.year = y; BirthDay.month = m; BirthDay.day = d; }
```

Такой конструктор прекрасно работает, однако лучше поступить в соответствии с принципом инкапсуляции: пусть инициализацией даты занимается конструктор структуры `Date`. Заодно, следуя все тому же принципу, внесем определение перечислимого типа `Month` внутрь структуры, т. к. больше этот тип нигде фактически не используется. Чтобы не писать в заголовках длинные имена типов, применим оператор `typedef`. Три конструктора — это наш обычный конструктор по умолчанию, конструкторы инициализации и копирования. Мы напишем еще и четвертый конструктор, в котором месяц инициализируется целым числом. Тогда со всеми конструкторами структура `Date` выглядит так (листинг 8.20).

Листинг 8.20. Структура `Date` с конструкторами

```
typedef unsigned int UI;
struct Date
{  enum Month
   { jan_=1, feb_, mar_, apr_, may_, jun_,
     jul_, aug_, sep_, oct_, nov_, dec_
   };
   UI year; Month month; UI day;     //  поля
   Date(){}                          //  конструктор по умолчанию
   Date(const Date &d)               //  конструктор копирования
   { year = d.year; month = d.month; day = d.day; }
    //  конструктор инициализации
   Date(const UI y, const Month m, const UI d)
   { year = y; month = m; day = d; }
    //  второй конструктор инициализации
   Date(const UI y, const UI m, const UI d)
   { if ((0 < m) && (m < 13)) { year = y; month = Month(m); day = d; } }
};
```

Во втором конструкторе инициализации мы наблюдаем две особенности:

1. При присваивании значения месяцу мы вынуждены выполнить явное преобразование типа, иначе программа не транслируется.
2. Конструктор выполняет проверку месяца и инициализирует поля только при правильных значениях. Присвоить нули мы не можем, поскольку тип `Month` не содержит такой константы. Как видим, у конструктора нет

возможности сообщить о результате выполнения инициализации. Это явилось одной из причин включения в C++ *механизма исключений*.

Мы прописали конструкторы внутри структуры, но можно реализовать их "внешним" образом, который для более сложных структур данных является основным. Форма записи "внешних" конструкторов такая:

```
Date::Date(const UI y, const Month m, const UI d)
{ year = y; month = m; day = d; }
Date::Date(const UI y, const UI m, const UI d)
{ if ((0 < m) && (m < 13)) { year = y; month = Month(m); day = d; } }
```

Испробуем инициализацию дат — работают все варианты, кроме одного — списка инициализации структуры. Но так прописано в стандарте: если в структуре нет конструктора, то разрешается список инициализации структуры; если конструктор есть, то список инициализации структуры запрещен. Возможны варианты:

```
Date A;                                  //  работает
Date B = {2001, Date::feb_, 22};         //  теперь не работает
Date C(2002, Date::feb_, 22);            //  вместо предыдущего
Date G(2003, 2, 22);                     //  4-й конструктор
Date D(B);                               //  работает
Date F = B;                              //  работает
Date H = Date(2005, Date::feb_, 22);     //  работает
Date E = Date(2007, 7, 29);              //  работает 4-й конструктор
```

Поскольку мы внесли определение `Month` внутрь структуры, то при использовании его констант системе требуется уточняющий квалификатор. Теперь работает конструктор структуры `Person`:

```
Person U("Это фамилия", Date(1986, Date::jul_, 29));
```

Как мы видим, на месте второго аргумента написан вызов конструктора `Date`, который создает анонимный объект.

Присвоение значений полям структуры не обязательно делать в *теле* конструктора. Специально для конструкторов инициализации в C++ придумали *список инициализации*[1]. Определим конструкторы структуры `Rational` (их текст приведен в листинге 8.21).

Листинг 8.21. Конструкторы типа `Rational`

```
typedef unsigned long ULONG
struct Rational
```

[1] Не путайте с обычным списком инициализации структуры, когда конструкторы не определены.

```
{  ULONG num, denum;
   Rational(){ num = 0; denum = 1; }     //  по умолчанию
   Rational(ULONG n)                     //  инициализации
   { num = n; denum = 1; }
   Rational(ULONG n, ULONG d)            //  инициализации
   { if (d != 0) { num = n; denum = d; } }
   Rational(const Rational &r)           //  копирования
   { num = r.num; denum = r.denum; }
};
```

Конструкторы со списком инициализации приведены в листинге 8.22.

Листинг 8.22. Конструкторы со списком инициализации

```
struct Rational
{  ULONG num, denum;
   Rational():num(0), denum(1){ }
   Rational(ULONG n) :num(n), denum(1){ }
   Rational(ULONG n, ULONG d): num(n), denum((d != 0)?d:denum) { }
   Rational(const Rational &r): num(r.num),denum(r.denum){ }
};
```

Как видим, тело конструкторов стало пустым. Обратите внимание, в третьем конструкторе (см. листинг 8.22) мы в списке инициализации используем выражения, проверяя знаменатель на ноль. Можно использовать и функции, например:

```
Rational(ULONG n, ULONG d):num(n), denum(init(d)) { }
```

Функция может быть определена в той же структуре, например, так:

```
ULONG init(ULONG y) { return (y != 0)?y:denum)); }
```

С точки зрения результата инициализации, такой конструктор идентичен определенному нами ранее с присвоением значений в теле, однако инициализация через список осуществляется до начала выполнения тела конструктора. Поэтому мы используем уже проинициализированные поля в теле конструктора. Кроме того, как мы помним, у нас были проблемы с объявлением константы внутри структуры. Список инициализации позволяет проинициализировать поле-константу.

Конструкторы и параметры функций

При наличии конструктора инициализации в качестве параметра допускается передавать анонимный объект-структуру (листинг 8.23).

Листинг 8.23. Анонимный объект в качестве параметра

```
struct Old { int a; float b;          //  поля
             Old(int x, float y)      //  инициализирующий конструктор
             { a = x; b = y; }
           };
Old g(Old p)                          //  параметр — структура
{ return p; }
void main(void)
{ Old x = g(Old(1, 2.0)); }           //  параметр — анонимный объект
```

Инициализация выполняется сразу при создании анонимного параметра. Без конструкторов такая запись попросту невозможна.

Если структура передается по указателю, то при наличии конструктора можно при вызове задавать даже динамический анонимный объект. Текст структуры приведен в листинге 8.24.

Листинг 8.24. Динамический анонимный объект в качестве параметра

```
struct Old { int a; float b;
             Old(int x, float y)      //  инициализирующий конструктор
             { a = x; b = y; }
           };
Old g(Old *p)                         //  параметр — указатель
{  p->a = p->b+6;                     //  доступ к элементу операцией "->"
   (*p).b = p->a+7;                   //  доступ к элементу селектором "."
   return *p;                         //  возврат значения, а не указателя
}
void main(void)
{ Old x = g(new Old(1, 2.0)); }       //  анонимная динамическая структура
```

Конструкторы и преобразование типов

Структуры позволяют нам объявлять новые типы, поэтому преобразование для новых типов тоже надо уметь делать. Роль преобразователя играют конструкторы. Вспомним объявление объекта типа `Person`:

```
Person B = Person("Привет большой и горячий!");
```

Фактически здесь происходит такое преобразование:

```
const char * -> Person
```

Вспомним (см. листинг 8.13 и 8.14), что при двух типах данных, один из которых новый, требуется написать не менее трех вариантов для каждой

операции. За счет преобразований типов можно значительно сократить количество перегружаемых функций — для этого в конструкторе необходимо использовать параметры по умолчанию. Этот же прием сократит нам и количество конструкторов — в этом случае нам достаточно будет написать только один конструктор, который и будет работать во всех случаях (его текст приведен в листинге 8.25).

Листинг 8.25. Конструктор преобразования

```
struct Rational
{  ULONG num, denum;
   Rational(ULONG n = 0, ULONG d = 1)
   { if (d != 0) { num = n; denum = d; } }
};
```

Теперь единственный конструктор выполняет функции всех трех. Кстати, два конструктора с параметрами по умолчанию писать нельзя — будет ошибка трансляции. Наличие такого конструктора обеспечивает нам преобразование типа по умолчанию из целого в рациональное число:

```
long -> Rational
```

Здесь допустимо такое объявление:

```
Rational a = 3;
```

Этот же конструктор позволяет нам написать единственную функцию-операцию вместо трех, которую можно использовать с любыми сочетаниями допустимых типов параметров. Рассмотрим реализацию операции сравнения == рациональных чисел. Операция бинарная, результатом является значение типа `bool`, поэтому реализация может быть следующей:

```
bool operator==(Rational a, Rational b)
{ return (a.num*b.denum == b.num*a.denum); }
```

В совокупности с определенным в структуре конструктором данная функция обеспечивает корректную работу всех вариантов сравнений. Обратите также внимание на то, что использование операции == в теле функции не является рекурсивным вызовом — в данном случае она работает со встроенным типом данных.

При наличии преобразования в одну сторону естественно иметь возможность выполнять и обратное преобразование:

```
Rational -> long
```

C++ позволяет это сделать — мы должны определить внутри структуры функцию преобразования:

```
operator long() { return (num/denum); }
```

Мы осуществили перегрузку функции-операции. При наличии такой функции внутри структуры становятся доступными присвоения объектов типа `Rational` целым переменным:

```
int main(void)
{  Rational mm = 11;       //  int -> long —> Rational неявное
   cout << mm << endl;     //  выводит (11/1)
   long i = mm;            //  Rational -> long неявное
   cout << i << endl;      //  выводит 11
   return 0;
}
```

Обратите внимание, что переменной `i` мы присвоили не поле структуры, а сам объект типа `Rational`. В данном случае неявно вызывается функция преобразования в тип `long`. Аналогичные неявные преобразования выполняются и при передаче параметров в функцию по значению. Пусть у нас определены функции:

```
Rational f(Rational a)
{ return a; }
long ff(Rational a)
{ return a; }
```

Тогда допускаются такие вызовы:

```
Rational zz = f(13);
long k = ff(zz);
```

В первом случае осуществляется неявное преобразование целого числа 13 в тип `Rational` конструктором, и переменная `zz` получает значение `(13/1)`. А во втором случае неявное преобразование в целое выполняется при возврате, и переменной `k` присваивается значение `13`.

Естественно, можно запретить неявные преобразования. Что касается преобразования `Rational -> long`, то это легко сделать, просто удалив определение функции преобразования из структуры. Для запрещения неявного преобразования `long -> Rational` в C++ существует специальное зарезервированное слово `explicit`. Заголовок конструктора инициализации должен быть таким:

```
explicit Rational(ULONG n = 0, ULONG d = 1)
```

Тогда допускается только явный вызов конструктора как при объявлении с инициализацией, так и при передаче параметра. Следующие простые примеры демонстрируют это:

```
Rational mm = Rational(11);
Rational zz = f(Rational(13));
```

Если не писать справа тип `Rational`, возникают ошибки трансляции.

А теперь вспомним неудачную попытку ввести для типа `double` операцию возведения в степень. Используя структуру, конструкторы и функции обратного преобразования, мы можем сделать это достаточно просто. Объявим структуру `double`, текст которой приведен в листинге 8.26.

Листинг 8.26. Тип-оболочка для `double`

```
struct Double
{  double t;
   Double (double a = 0.0): t(a) { }
   operator double() { return t; }
}
```

После этого достаточно перегрузить необходимые операции для работы с типом `double`. Мы не будем писать их все, поскольку они простые, реализуем только возведение в целую степень:

```
double operator^(const Double &a, int b)
{  double t = a;                          //  double <- Double
   if (b == 0) return 1;                  //  int -> double
   else if (t == 0) return 0;             //  int -> double
   else if (b == 1) return a;             //  Double -> double
   else if (b < 0) { t = 1/t; b =- b; }   //  степень отрицательная
   double p = 1;
   for (int i = 1; !(i > b); i++) p *= t; //  вычисляем степень
   return p;
}
```

Проверка показывает, что функция работает корректно:

```
int main(void)
{  Double d = 0;
   cout << double(d^-2) << endl;    //  0^-2 = 0
   cout << double(d^0) << endl;     //  0^0 = 1
   d = 2;                           //  корректно работает
   cout << double(d^0) << endl;     //  2^0 = 1
   cout << double(d^1) << endl;     //  2^1 = 2
   cout << double(d^-2) << endl;    //  2^-2 = 0.25
   cout << double(d^3) << endl;     //  2^3 = 8
   return 0;
}
```

Здесь надо обратить внимание на несколько моментов. Во-первых, при объявлении и инициализации переменной `d` дважды работает неявное преобразование: стандартное `int -> double` и определенное нами `double -> Double`.

Во-вторых, корректно работает операция присваивания, очевидно выполняя те же два преобразования, хотя мы эту операцию не перегружали. В-третьих, явное указание преобразования в тип `double` при выводе нам потребовалось только потому, что не была перегружена операция вывода для `Double`.

Таким образом, мы можем легко ввести "новые" арифметические операции для любого встроенного типа данных. Собственно, мы действовали аналогично разработчикам языка программирования Java, реализовав тип-оболочку для встроенного типа данных.

Перегрузка операций методами

Перегрузка операций для структуры `Rational` внешними функциями нарушает принцип инкапсуляции. Раз уж структура определяет новый тип, то естественно "поплотнее" связать операции с типом. Нужно реализовать операции как методы структуры. Начнем с бинарной операции `+=`, поскольку проблемы ее реализации нам хорошо знакомы. Затем реализуем операцию присваивания, поскольку она может быть реализована только "внутренним" образом, и операцию инкремента, т. к. она является унарной.

Вспомогательные функции `gcd` и `reduce` тоже можно внести в структуру. Реализуем `reduce` "внешним" образом, а `gcd` прямо внутри структуры (текст приведен в листинге 8.27).

Листинг 8.27. Вспомогательные функции структуры `Rational`

```
void Rational::reduce()
{  ULONG t = ((num > denum)? gcd(num, denum):gcd(denum, num));
   num /=t; denum /=t;
}
ULONG gcd(ULONG x, ULONG y)
{ if (y == 0) return x; else return gcd(y, x%y); }
```

Функция `reduce` теперь параметров не имеет, т. к. функция-метод имеет доступ к любым элементам структуры. Однако в функции `gcd` мы оставили параметры, чтобы не изменять значения полей числителя и знаменателя при вычислении наибольшего общего делителя.

Теперь займемся реализацией операции `+=`. Попытаемся просто перенести операцию `+=` (см. листинг 8.10) внутрь структуры `Rational`. Вспомним, что заголовок операции имеет вид:

```
Rational& operator+=(Rational &a, const Rational &b)
```

Однако при трансляции тут же выясняется, что мы определили операцию неверно:

```
error C2804: binary 'operator +=' has too many parameters
бинарная 'операция +=' имеет слишком много параметров
```

Сообщение сбивает с толку, т. к. бинарная операция должна иметь два параметра, что мы и наблюдаем в заголовке. Однако при перегрузке бинарной операции функцией-методом это является ошибкой — один параметр у такой функции всегда определен по умолчанию. Любая функция-метод "знает", для какого объекта она вызвана. Вспомним, как вызывается функция-метод структуры:

```
Имя_переменной.имя_метода(параметры)
```

`Имя_переменной` и определяет этот текущий объект и передается функции-методу как неявный параметр. Пусть у нас объявлена переменная `t` типа `Rational`, и нам нужно увеличить ее на `3`. Тогда функциональная запись вызова операции-метода будет такой:

```
Rational t = 2;
t.operator+=(3)
```

Переменная `t` является первым операндом нашей операции. Собственно, обычная инфиксная форма записи является сокращением функциональной — удалено сочетание `.operator`. Функциональная запись однозначно показывает, что левый операнд — всегда объект определяемого типа. Любая функция-метод может ссылаться на него, используя ключевое слово `this`. Это указатель, поэтому для доступа к самому текущему объекту или его элементам надо использовать одну из операций доступа к структуре по указателю: либо `*`, либо `->`.

Таким образом, при объявлении операции `+=` как метода параметр у нее должен быть один. Заголовок тогда становится следующим:

```
Rational& operator+=(const Rational &b)
```

Это относится ко всем бинарным операциям.

Разберемся теперь с результатом. Изменяться должен первый аргумент, т. е. текущий объект. Кроме того, функция должна возвращать ссылку. Таким образом, по аналогии с внешней операцией (см. листинг 8.10) реализация операции `+=` должна быть такой (ее текст приведен в листинге 8.28).

Листинг 8.28. Операция-метод с присваиванием

```
Rational& Rational::operator+=(const Rational &b)
{  num = num*b.denum + b.num*denum; denum = denum*b.denum;
   (*this).reduce();     //  доступ к элементу объекта
   return (*this);       //  возврат текущего объекта
}
```

Мы реализовали метод вне структуры. Вызов функции `reduce` можно выполнить и так:

```
this -> reduce();
```

Возвращается `(*this)`, а не `this`, т. к. требуется вернуть ссылку, а не указатель (вспомним "левые" функции — листинг 7.5).

Испробуем нашу операцию. Исходная операция `+=`, встроенная в C++, является правоассоциативной и позволяет многократные присваивания. Поэтому реализуем операцию с несколькими переменными, а для сравнения выполним те же действия с целыми числами:

```
Rational t, z;
int it, iz;
t = 2, z = 3; t += z += 2;         //  результат t = (7, 1), z = (5, 1)
it = 2, iz = 3; it += iz += 2;     //  результат тот же
t = 2, z = 3; t+=(z += 2);         //  результат t = (7, 1), z = (5, 1)
it = 2, iz = 3; it+=(iz += 2);     //  результат тот же
t = 2, z = 3; (t += z)+=2;         //  результат t = (7, 1), z = (3, 1)
it = 2, iz = 3; (it += iz)+=2;     //  результат it = 7, iz = 3
```

Как мы видим, при "внутренней" реализации, как и при "внешней", возврат ссылки обеспечивает правильное выполнение операции.

Теперь давайте перегрузим операцию присваивания, которая тоже является многократной и правоассоциативной, но ничего не прибавляет к своему левому аргументу. Реализуем операцию `=` по аналогии с операцией `+=`. Текст реализации приведен в листинге 8.29.

Листинг 8.29. Операция присваивания

```
Rational& Rational::operator=(Rational b)
{ num = b.num; denum = b.denum; return *this; }
```

Проверка показывает:

```
t = 2; z = 3; (t = z) = 7;         //  результат t = 7, z = 3
it = 2; iz = 3; (it = iz) = 7;     //  результат it = 7, iz = 3
```

что наша операция является многократной. Именно для реализации перегрузки операций присваивания Б. Страуструп включил в C++ ссылки. Ну, а потом уже оказалось, что ссылки полезны и в других конструкциях, в частности при передаче параметров. Нужно сказать, что любой метод, если необходимо, может возвращать ссылку. По примеру Б. Страуструпа [37] добавим в структуру `Date` (см. листинг 8.20) методы, увеличивающие год, месяц и день на 1 (их текст приведен в листинге 8.30).

Листинг 8.30. Методы, возвращающие ссылку

```
Date& AddYear (int n) { year += n; return *this; }
Date& AddMonth (int n) { month = Month(month + n); return *this;}
Date& AddDay (int n) { day += n; return *this;}
```

Допустим, у нас еще определена операция вывода даты <<. Тогда для увеличения всех полей переменной

```
Date d(22, 02, 2003);
```

мы можем написать вызов всех трех функций сразу в одном выражении:

```
cout << d.AddYear(1).AddMonth(1).AddDay(1) << endl;
```

На экране получим:

```
23 3 2004
```

Вернемся к нашей структуре `Rational`. Реализуем теперь операцию инкремента. Поскольку операция одноместная, при ее реализации как функции-метода параметров быть не должно. Префиксная операция выглядит так:

```
Rational Rational::operator++()
{ return (*this += 1); }
```

Постфиксная операция, как и при перегрузке внешней функцией, должна иметь фиктивный целый параметр. Мы не используем этот параметр, поэтому и не указываем его имя:

```
Rational Rational::operator++(int)
{ Rational t = *this; *this += 1; return t; }
```

Обратите внимание на то, что мы внутри функции-метода объявили другой объект того же типа и присвоили ему "текущий" объект. Протестируем наши, вновь определенные функции, с помощью следующей программы:

```
int main(void)
{  Rational z = 2;
   cout << ++z << endl;     //  выводит (3/1)
   cout << z++ << endl;     //  выводит (3/1)
   cout << z << endl;       //  выводит (4/1)
}
```

Совершенно аналогично можно перегрузить операции вычитания с присвоением -= и декремента -- .

Внешняя или внутренняя?

Попробуем теперь реализовать перегрузку "простой" операции сложения как функции-метода. Рассмотрим сложение рационального числа с целым числом. Текст реализации приведен в листинге 8.31.

Листинг 8.31. Реализация метода-операции сложения

```
Rational operator+(int b)
{ Rational t = *this; return (t += b); }
```

Однако при проверке данной операции

```
Rational t = 2;
cout << t + 3 << endl;     //  выводит (5, 1)
cout << 3 + t << endl;     //  выводит 5
```

вдруг выясняется, что второй вариант имеет результатом не рациональное, а целое число — результат изменился от перемены мест слагаемых! Представим последние два вызова функции сложения в функциональной форме:

```
cout << t.operator+(3) << endl;
cout << 3.operator+(t) << endl;
```

Если первый оператор совершенно правилен и в точности соответствует нашему определению, то второй просто не транслируется — Visual C++ 6 выдает аж четыре сообщения об ошибках. Такой оператор недопустим, поскольку число 3 не является объектом типа `Rational`. Инфиксная форма работает, поскольку у нас определена функция преобразования в базовый тип `long`. Если эту функцию закомментировать, то при трансляции также выдается несколько сообщений об ошибках. Чтобы получить результат типа `Rational`, мы должны прописать явное преобразование типа так:

```
cout << Rational(3+t) << endl;     //  выводит (5, 1)
```

либо так:

```
cout << Rational(3)+t << endl;     //  выводит (5, 1)
```

Таким образом, перегрузка операции сложения посредством функции-метода не является коммутативной при разнотипных операндах. Как мы помним, "внешняя" перегрузка как для `Decade` (см. листинг 8.5), так и для `Rational`, таких проблем не создавала. Поэтому знатоки C++ рекомендуют перегружать такие функции внешними функциями.

Необходимо, тем не менее, отметить, что операция сложения с двумя аргументами типа `Rational` (несмотря на то, что мы не определяли такую операцию — преобразование делает конструктор) является коммутативной и работает совершенно правильно в обоих вариантах:

```
Rational t = 2, z = 3;
cout << t + z << endl;     //  выводит (5, 1)
cout << z + t << endl;     //  выводит (5, 1)
```

Операцию вывода `<<` мы ранее тоже всегда определяли как внешнюю. Теперь попробуем выполнить перегрузку операции вывода функцией-методом.

В качестве примера рассмотрим определенную ранее внешним образом функцию вывода рациональных чисел (см. листинг 8.12). У внешней функции было два параметра: объект вывода и выводимый объект. У функции-метода, очевидно, должен быть один явный параметр — поток вывода, поскольку выводимый объект передается неявно. Таким образом, возможная реализация выглядит так (ее текст приведен в листинге 8.32).

Листинг 8.32. Вывод как метод

```
ostream& operator<<(ostream& t)
{ return (t << '(' << num << '/' << denum << ')'); }
```

В определении ничего необычного не наблюдается. Однако напишем функциональный вызов такой функции для некоторого объекта `R`:

```
R.operator<<(cout);
```

В сокращенной записи получим:

```
R<<cout;
```

Операция вывода выглядит как операция ввода — запись, совершенно сбивающая с толку! Согласитесь — это совсем не то, что мы хотели. Такая ситуация возникает потому, что "текущий" объект всегда является левым аргументом функций-методов. Для операций с присваиванием это прекрасно подходит, для других — далеко не всегда. Именно поэтому операции ввода/вывода перегружаются практически всегда внешними функциями — чтобы не нарушался традиционный вид операций.

Таким образом, полное определение нашего нового типа рациональных чисел в настоящий момент приведено в листинге 8.33.

Листинг 8.33. Определение типа `Rational`

```
typedef unsigned long ULONG;
struct Rational
{  ULONG num, denum;
   Rational(ULONG n = 0, ULONG d = 1):num(n), denum(d) { reduce(); }
   Rational& operator+=(const Rational &b);
   Rational& operator=(Rational b);
   Rational operator++();
   Rational operator++(int);
   operator long() { return num/denum; }
   ULONG gcd(ULONG x, ULONG y)
   { if (y == 0) return x; return gcd(y, x%y); }
   void reduce();
};
```

```
void Rational::reduce()
{  ULONG t = ((num > denum)? gcd(num, denum):gcd(denum, num));
   num /=t; denum /=t;
}
Rational& Rational::operator+=(const Rational &b)
{  num = num*b.denum + b.num*denum;
   denum = denum*b.denum;
   (*this).reduce();
   return *this;
}
Rational& Rational::operator=(Rational b)
{ num = b.num; denum = b.denum; return *this; }
Rational Rational::operator++()
{ return (*this += 1); }
Rational Rational::operator++(int)
{ Rational t = *this; *this += 1; return t; }
//-------------- внешние функции -----------------
Rational operator*(Rational a, Rational b)
{  Rational c;
   c.num = a.num*b.num;
   c.denum = a.denum*b.denum;
   c.reduce();
   return c;
}
Rational operator+(Rational a, Rational b)
{ Rational t = a; return (t += b); }
bool operator == (Rational a, Rational b)
{ bool t = (a.num*b.denum != b.num*a.denum); return (!t); }
ostream& operator<<(ostream& t, Rational d)
{ (t << '(' << d.Num() << '/' << d.Denum() << ')'); return t; }
istream& operator>>(istream& t, Rational& a)
{ t >> a.n >> a.d; return t; }
```

Классы и структуры

Настала пора познакомиться с настоящими классами, свойства которых мы обсуждали на примере структур. В конце концов, именно классы стали первой отличительной чертой C++ от C. Конструкция класса по внешнему виду отличается от структуры только одним словом:

```
class имя_класса
{ поля и методы класса };
```

Поля и методы объявляются для класса абсолютно точно так же, как мы это делали для структур. Тогда возникает законный вопрос: зачем уважаемый создатель C++ добавил "лишнее" ключевое слово? Чтобы разобраться, давайте в листинге 8.32 заменим слово `struct` на слово `class` и попробуем выполнить простую программу:

```
int main()
{ Rational A = 1, B = 2; cout << A << endl; return 0; }
```

Нас ждет разочарование — программа перестала работать. Выдается несколько одинаковых сообщений об ошибках `C2248` вида:

```
'Rational::Rational': cannot access private member declared in class
```

Смысл таков, что невозможен доступ к конструктору, поскольку он является "личной собственностью" класса `Rational`. Доступ — закрыт! Очевидно, что такое решение было принято Б. Страуструпом в противовес абсолютной открытости структуры. Он не мог просто "закрыть" структуру, поскольку изначально была продекларирована совместимость с C, а в C структура открыта. Поэтому и было принято решение создать новую конструкцию, в которой реализован принцип инкапсуляции. Но тогда потребовалось добавить и другие ключевые слова, с помощью которых видимостью для внешних клиентов элементов класса можно управлять. Ключевое слово `public` открывает доступ, а слово `private` — закрывает.

Давайте перепишем структуру `Rational`, разграничив доступ к элементам структуры. Здесь сразу надо сказать, что конструкторы должны быть открыты, иначе мы не сможем создавать переменные типа `Rational`. А вот доступ к остальным элементам оставим на усмотрение разработчика класса. Обычно элементы данных все-таки закрываются — в соответствии с принципом инкапсуляции. Хорошими "кандидатами" в закрытые члены являются и наши вспомогательные функции, т. к. они выполняют "чисто внутреннюю" работу для типа `Rational`.

Поскольку структура по умолчанию открыта, нам нет необходимости явно прописывать слово `public`, поэтому структура `Rational` с разграничением доступа выглядит так (ее текст приведен в листинге 8.34).

Листинг 8.34. Структура `Rational` с разграничением доступа

```
struct Rational
{  Rational(ULONG n = 0, ULONG d = 1):num(n), denum(d) { reduce(); }
   Rational& operator+=(Rational &b);
   Rational& operator=(Rational b);
   Rational operator++();
   Rational operator++(int);
```

```
   operator long() { return num/denum; }
   private:
   ULONG num, denum;
   ULONG gcd(ULONG x, ULONG y)
   { if (y == 0) return x; return gcd(y, x%y); }
   void reduce();
};
```

Эквивалентный класс `Rational` приведен в листинге 8.35.

Листинг 8.35. Класс `Rational`

```
class Rational
{  public:
   Rational(ULONG n = 0, ULONG d = 1):num(n), denum(d) { reduce(); }
   Rational& operator+=(Rational &b);
   Rational& operator=(Rational b);
   Rational operator++();
   Rational operator++(int);
   operator long() { return num/denum; }
   private:
   ULONG num, denum;
   ULONG gcd(ULONG x, ULONG y)
   { if (y == 0) return x; return gcd(y, x%y); }
   void reduce();
};
```

И в классе, и в структуре можно написать столько слов `public` и `private`, сколько нам необходимо, и в том порядке, как требуется. Очередное слово действует до следующего. В принципе, если хотим, мы можем каждый элемент класса индивидуально объявлять либо открытым, либо закрытым. Открытая часть класса по традиции называется *интерфейсом*.

Методы класса имеют неограниченный доступ к полям класса, независимо от того, закрыты поля или нет. А вот с внешними функциями придется разобраться. Попробуем выполнить простую программу:

```
int main()
{ Rational A = 1, B = 2; cout << A + B << endl; return 0; }
```

Неожиданно оказывается, что программа не транслируется — внешние функции не имеют доступа к закрытым элементам класса. Одно решение этой проблемы нам известно: сделать все функции методами класса. Но это неудобно — мы помним, что реализация некоторых операций как методов приводит к интересным эффектам. Б. Страуструп, очевидно, тоже столкнул-

ся с такими проблемами, поэтому он их решил, включив в язык механизм "друзей". "Друзья" — это внешние функции или другие классы, которые имеют доступ к закрытым частям нашего класса. Поэтому мы должны объявить все внешние функции "друзьями", как приведено в листинге 8.36. Это делается в интерфейсе класса посредством слова `friend`.

Листинг 8.36. Интерфейс класса `Rational`, версия 2

```
typedef unsigned long ULONG;
class Rational
{  public:
   Rational(ULONG n = 0, ULONG d = 1):num(n), denum(d) { reduce(); }
   Rational& operator+=(Rational &b);
   Rational& operator=(Rational b);
   Rational operator++();
   Rational operator++(int);
   operator long() { return num/denum; }
   friend bool operator==(Rational a, Rational b);
   friend ostream& operator<<(ostream& t, Rational r);
   friend istream& operator>>(istream& t, Rational& a);
   friend Rational operator*(Rational a, Rational b);
   friend Rational operator+(Rational a, Rational b);
   private:
   ULONG num, denum;
   ULONG gcd(ULONG x, ULONG y)
   { if (y == 0) return x; else return gcd(y, x%y); }
   void reduce();
};
```

После этого все работает. Таким образом, мы реализовали то, что Б. Страуструп называет "конкретными типами" [37]. Объекты таких типов подчиняются всем правилам, которые применяются к встроенным типам.

ГЛАВА 9

Динамические классы

Использование указателей и динамических переменных в классах в сочетании с перегрузкой операций составляет удивительно мощный механизм создания новых типов данных. Собственно, вся библиотека STL реализована с использованием именно этих свойств C++. Однако разрабатывать классы с указателями необходимо особенно тщательно. На этом поприще нас "под каждым кустом" подстерегает самая коварная ошибка — утечка памяти. Противодействие этому составляет львиную долю работы программиста. Но при должной аккуратности выгоды от реализации динамических классов велики: значительно упрощается реализация типичных ситуаций, возникающих при программировании многих задач. Примером могут служить контейнеры стандартной библиотеки.

Массивы с задаваемыми индексами

Как известно, в Pascal при объявлении массива нужно задавать нижний и верхний индексы массива. При переходе на C++ "паскалисты" с трудом привыкают к отсутствию индексов в объявлениях массивов. Чтобы облегчить такой переход, а заодно разобраться в нюансах взаимоотношений классов и динамических переменных, перегрузки операций индексирования и присваивания, реализуем новый тип — массив с задаваемыми нижним и верхним индексом.

Разрабатывая новый класс, мы обычно начинаем с определения интерфейса, который мы хотим предоставить пользователям (клиентам) этого класса.

Примечание

Под клиентами мы будем понимать программу, в которой этот класс используется.

Название нового типа пусть будет `array`. Рассмотрим, в первую очередь, возможные конструкторы новых массивов. Один конструктор очевиден — аргументами являются нижний и верхний индексы элементов массива:

```
array (const int left, const int right);
```

Если добавить к индексам инициализирующее значение, то получим второй конструктор. Третий конструктор — конструктор копирования; аргумент — объект типа `array`. Новый массив получает значения из массива-аргумента. Еще один вариант — вырезка их объекта типа `array`, задаваемая двумя индексами. По примеру STL реализуем конструктор нашего массива из встроенного, с двумя указателями: на начало и конец последовательности элементов. И в этом случае новый массив заполняется значениями из массива-инициализатора. Реализуем еще традиционный конструктор с одним аргументом — количеством элементов. В этом случае нижний индекс, очевидно, равен 0, а верхний на 1 меньше количества. Таким образом, у нас получается 6—7 конструкторов. Можно немного сократить это количество за счет применения параметров по умолчанию.

Операция индексирования `[]`, естественно, должна быть перегружена. Так как конструкция `имя[индекс]` может появляться слева и справа от знака присваивания, операция должна возвращать ссылку. Эта операция должна проверять индекс на допустимость.

Реализуем и операцию присваивания, чтобы иметь возможность присваивать наши массивы друг другу. Однако операция присваивания сродни конструктору копирования — в операции участвует весь массив. Нам же, очевидно, могут потребоваться вырезки, для которых надо задавать индексы первого и последнего копируемого элемента. В операции присваивания мы не можем это сделать, т. к. операция бинарная и не может иметь более двух аргументов, один из которых текущий объект. Поэтому, по примеру STL, реализуем функцию-метод `assign`.

Операции, в которых одним из аргументов является скалярная величина, а вторым — наш массив (например, умножение на константу) тоже лучше реализовать как операции с присваиванием. Очень полезно также иметь операции, выдающие количество элементов массива, нижний и верхний индексы.

Раз уж мы реализуем новый класс, давайте расширим набор операций: добавим методы, которые вычисляют максимальный и минимальный элемент, сумму и произведение элементов массива. В стандартной библиотеке для любого контейнера реализованы методы сортировки и поиска элемента — мы поступим так же. Можно реализовать много различных операций, но ограничимся операциями с двумя массивами: найдем поэлементные суммы, произведения и скалярное произведение, которое может нам понадобиться при реализации класса матриц.

Кстати, некоторые операции вроде сортировки или поиска максимума являются унарными по своей природе. Поэтому мы вполне можем перегрузить любую унарную операцию, написав тело соответствующей функции. Например операция ~ (битовое отрицание) может у нас сортировать массив, а операция ! — вычислять максимум. Согласитесь, красиво выглядит:

```
~a;    //  сортировка массива
```

Давайте с сортировкой так и поступим, а остальные операции реализуем традиционным способом, задавая имя функции. Надо только отследить, чтобы имена наших более-менее сложных функций не совпадали со стандартными, иначе мы не сможем использовать стандартные алгоритмы.

Мы ни слова до сих пор не говорили о типе элементов массива. Однако некоторые наши операции (например, сумма или произведение элементов массива) предполагают, что тип элементов числовой. Реализуем массив типа `double`. Интерфейс нашего класса приведен в листинге 9.1.

Листинг 9.1. Интерфейс класса — массив с индексами

```
class array
{  public:
   typedef unsigned int UINT;
   array(UINT size = 10, double k = 0.0);
   array(int l, int r, double k = 0.0);
   array(const array &a);
   array(const array &a, int l, int r);
   array(const double * const begin, const double * const end);
   ~array();
   double& operator[](int index);
   array& operator=(const array &a);
   array& assign(const array &a, int left, int right);
   array& assign(const double *begin, const double *end);
   array& operator+=(const double &a);
   array& operator-=(const double &a);
   array& operator*=(const double &a);
   array& operator/=(const double &a);
   int Left()const;           //  выдать левый индекс
   int Right()const;          //  выдать правый индекс
   UINT size()const;          //  выдать количество элементов
   double& max_value();
   double& min_value();
   double summa();
   double product();          //  произведение элементов
   void operator~();          //  сортировка
   double* find(double a);    //  поиск
   friend array operator+(const array&a, const array &b);
```

```
   friend array operator*(const array&a, const array &b);
   friend double product_inner (const array&a, const array &b);
   friend ostream& operator <<(ostream& to, const array &a);
   friend istream& operator >>(istream& to, array &a);
   private: double* data;       //  динамический массив
            double dummy;       //  фиктивный элемент
            int left, right;    //  нижний и верхний индексы
            UINT size_array;    //  количество элементов
};
```

В листинге мы прописали оператор `typedef` — это означает, что наш класс предоставляет клиенту имя `UINT` как синоним типа `unsigned int`. Функции `min`, `max` возвращают ссылки, чтобы можно было их использовать слева от присваивания. Функция `find_value` возвращает указатель на найденный элемент, а если такого элемента в массиве нет, возвращает `0`.

Конструкторы и деструкторы

Сначала реализуем конструкторы. Каждый из них создает динамический массив, доступ к которому выполняется по закрытому указателю `data`. Однако выделенную память надо возвращать, иначе возникает утечка памяти. Доступа к указателю у клиента нет, поскольку указатель находится в закрытой части. Да и не нужно доступ клиенту давать — этим мы грубо нарушим принцип инкапсуляции. Возврат памяти должен делать наш класс. Это можно было бы делать явным образом, предоставив пользователю функцию-метод вроде `Destroy` (разрушить), в которой вызывается операция `delete[]`. Однако такой подход мало чем отличается от явного задания `delete[]`: клиент должен сам задавать уничтожение объекта. В C++ есть стандартный механизм, предназначенный как раз для таких целей — деструктор. Конструктор создает объект, деструктор его уничтожает.

Деструктор — это функция, имеющая имя класса, но первым символом должен быть символ `~` (тильда). Возвращаемого значения у деструктора нет, а параметров — не должно быть. Деструктор вызывается автоматически при каждом уничтожении объекта. Например, для локальных объектов — при выходе из блока, для динамических — при каждом вызове `delete`. Деструктор может быть только один — перегрузка деструкторов не разрешается. В качестве примера покажем реализацию пары наиболее сложных конструкторов и деструктора. Текст реализации приведен в листинге 9.2.

Листинг 9.2. Реализация конструкторов и деструктора

```
array::array(const double * const begin, const double * const end)
{  if (begin < end)
```

```
   {  size_array = (end - begin);          //  количество элементов
      left = 0; right = size_array - 1;    //  индексы
      data = new double[size_array];
      copy(begin, end, data);
   }
}
array::array(const array &a, int l, int r)
{  if ((l < r) && (a.left <= l) && (r <= a.right))
   {  left = l; right = r; size_array = (right - left) + 1;
      data = new double[size_array];
      copy(a.data + (l - a.left),
           a.data + (l - a.left) + size_array, data);
   }
}
array::~array(){ delete[]data; data = 0; }
```

Остальные конструкторы более просты, поэтому читателю не составит труда самому реализовать их.

Деструктор короткий, поэтому лучше его реализовать как inline-функцию прямо в классе. Возникает естественный вопрос: почему мы не писали деструкторы раньше (см. листинг 8.36)? Потому что раньше мы в конструкторах не выполняли никаких действий, которые требуют "ручной уборки". Если деструктор не написан, система создает стандартный деструктор по умолчанию, который вызывается при уничтожении объекта. Б. Страуструп пишет [37], что такие деструкторы "ничего не делают". В данном случае автоматический деструктор не годится, т. к. он не вызывает операцию `delete` — это мы должны делать сами явным образом. Деструктор надо определять всякий раз, когда мы получаем какие-нибудь ресурсы в конструкторе. Например, конструктор может открывать файл, а деструктор — закрывать его.

При наличии определенных таким образом конструкторов мы можем создавать объекты типа `array` следующими способами:

```
double a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
array A;                    //  "пустой" массив double[10]
array B(a, a + (sizeof(a)/sizeof(double)));    //  из массива
array D(a + 3, a + 7);      //  из массива
array C(-2, +5);            //  индексы
array F(-5, +5, 2.2);       //  индексы с инициализацией
array G(B);                 //  другой массив
array X(F,0,5);             //  другой array с индексами
array Z(20);                //  количество
array W(10u, -1.0);         //  количество с инициализацией
```

Если в последнем объявлении первый аргумент мы зададим без суффикса u (просто 10), то получим неоднозначность вызова (конфликт с конструктором, имеющим параметры-индексы). При перегрузке необходимо строго соблюдать типы параметров во избежание возможных проблем.

Мы реализовали только "правильное" поведение конструкторов. Поэтому список инициализации может быть использован только в конструкторе копирования — там значения полей правильные. В остальных случаях надо сначала проверять параметры, а потом уже заполнять поля. Однако при объявлении объекта вполне возможно неправильное задание параметров (например, неправильно заданы индексы). Такие случаи конструктор обязан обрабатывать, однако сделать это он может только с помощью механизма исключений, который мы изучим в *гл. 10*. Аналогичные проблемы возникают и при реализации других операций, в которых необходимо оперировать индексами. Текст реализации операции индексирования приведен в листинге 9.3.

Листинг 9.3. Реализация операции индексирования

```
double& array::operator[](int index)
{  if (left <= index && index <= right) return data[i];
   else return dummy;
}
```

Можно, конечно, выдавать сообщение об ошибке и заканчивать работу, но такое решение для настоящих программных продуктов совершенно неприемлемо — гораздо лучше предоставить клиенту механизм обработки его ошибок. Пока мы не умеем им пользоваться, поэтому приходится возвращать "фиктивный" элемент (который тоже надо как-то инициализировать и отличать от допустимых элементов массива).

Копирующее присваивание

Динамическое выделение памяти в конструкторах требует от нас тщательной проработки операции присваивания. В динамическом классе мы не можем полагаться на автоматическую операцию, которую по умолчанию создает система. "Системное" присваивание копирует поля, а это совсем не то, что нам нужно. В самом деле, пусть у нас есть два массива `A` и `B`. В каждом из них свой указатель `data` указывает на собственный динамический массив. При выполнении присваивания `A = B` указатель `B.data` скопируется в `A.data` — получим потерянную ссылку! А нам требуется совсем другое: создать динамический массив размера `B.size_array`, скопировать туда значения массива `B`, а прежний динамический массив удалить. Кроме того, в случае `A = A` (присваивание самому себе) такие действия делать нельзя, а

поэтому при реализации операции копирующего присваивания (ее текст приведен в листинге 9.4) это надо отслеживать.

Листинг 9.4. Копирующее присваивание

```
array& array::operator=(const array &t)
{  if (this != &t)        //  отслеживаем присваивание самому себе
   {  double *new_data = new double[t.size_array];
      copy(t.data, t.data + t.size_array, new_data);
      delete[] data;      //  вернули память
      data = new_data;    //  вступили во владение
      left = t.left;
      right = t.right;
      size_array = t.size_array;
   }
   return *this;
 }
```

Однако мы можем последовать примеру Герба Саттера [35]: реализуем функцию обмена текущего массива с массивом-параметром, используя для этого функцию `swap` стандартной библиотеки, и с помощью нее — операцию присваивания. Текст реализации приведен в листинге 9.5.

Листинг 9.5. Операция присваивания по Саттеру

```
void Swap(array &other)
{  swap(data, other.data);
   swap(left, other.left);
   swap(right, other.right);
   swap(size_array, other.size_array);
}
array& array::operator=(const array &t)
{ array temp(t); Swap(temp); return *this; }
```

Функцию обмена можно явно прописать в приватной части. Проверка присваивания самому себе здесь не нужна: просто поля обмениваются друг с другом, и ошибок не возникает.

Совершенно аналогично реализуется операция `assign` (ее текст приведен в листинге 9.6).

Листинг 9.6. Операция `assign` по Саттеру

```
array& array::assign(const array &t, int l, int r)
{  array temp(t, l, r);
```

```
   Swap(temp);
   return *this;
}
array& array::assign(const double *begin, const double *end)
{  array temp(begin, end);
   Swap(temp);
   return *this;
}
```

Остальные операции присваивания особых вопросов не вызывают, поскольку модифицируют текущий объект. Для полноты картины давайте приведем реализацию функций минимума, поиска и сортировки. Текст реализации приведен в листинге 9.7. Воспользуемся стандартными алгоритмами.

Листинг 9.7. Функции-методы класса `array`

```
void array::operator~()
{ sort(data, data + size_array); }
double& array::min_value()
{ return *min_element(data, data + size_array); }
double *array::find_value(double v)
{ double *i = find(data, data + size_array, v);
if (I != data + size_array) return i;
else return 0;
}
```

Осталось реализовать функции — "друзей". Для примера покажем реализации поэлементного произведения, скалярного произведения и функции вывода. Функция поэлементного произведения, текст которой приведен в листинге 9.8, проверяет совпадение размеров массивов: если размеры не совпадают, то возвращается первый массив.

Листинг 9.8. Реализация функции —"друга"

```
array operator*(const array &a, const array &b)
{  if (a.size() == b.size())
   {  array z(a.size());
      for (int i = 0; i < a.size(); ++i) z[i] = a[i] * b[i];
      return z;
   }
   else return a;
}
```

Однако тут нас подстерегает очередной сюрприз: функция не транслируется, т. к. операция индексирования не определена для константных масси-

вов. Добавляем соответствующую операцию, текст которой приведен в листинге 9.9.

Листинг 9.9. Константная операция индексирования

```
double array::operator[](int index) const
{  if (left <= index && index <= right)
   return data[index];
   else return dummy;
}
```

Проверив выполнение функции поэлементного произведения в отладочном режиме, можно убедиться, что для массива `z` используется не константный вариант, а с элементами массивов `a` и `b` работает новая константная функция.

Скалярное произведение и операция вывода при помощи STL реализуются совсем просто и приведены в листинге 9.10.

Листинг 9.10. Скалярное произведение и вывод

```
ostream& operator <<(ostream& to, const array &a)
{  ostream_iterator<double, char> out(to, " ");
   copy(a.data, a.data + a.size_array, out);
   return to;
}
double product_inner(const array &a, const array &b)
{  if (a.size() == b.size())
   {  double z = inner_product(a.data, a.data+a.size_array, b.data, 0.0);
      return z;
   }
}
```

Функция скалярного произведения тоже проверяет соответствие размеров, и тут мы сталкиваемся с той же проблемой, что и в операции индексирования: что делать, если размеры не совпадают? Пока отложим этот вопрос до *гл. 11*.

"Разноликие" матрицы

Вслед за одномерным массивом можно реализовать класс `Matrix`, воплощающий математическое понятие матрицы. Возможно очень много различных реализаций такого класса. Классический способ — динамический двумерный массив *(см. гл. 4)*. Пусть матрица задается количеством строк и

столбцов. Ничего принципиально нового в реализации конструкторов и деструкторов, операций обработки, операций присваивания по сравнению с реализацией класса `array` нет, поэтому приводить полный интерфейс класса нет необходимости. Приведем только реализацию одного из конструкторов и деструктора (ее текст приведен в листинге 9.11).

Листинг 9.11. Класс `Matrix`

```
class Matrix
{  public:
   Matrix(int n = 1, int m = 1, double k = 0.0);     //  конструктор
   ~Matrix();                                        //  деструктор
    //  методы класса
   private:
   double **data;     //  данные матрицы
   int n; int m;      //  строки и столбцы
};
    //  конструктор выделяет память
Matrix::Matrix(int n, int m, double k):n(n), m(m)
{  int i, j;
   data = new double*[n];
   for (i = 0; i < n; ++i) data[i] = new double[m];
   for (i = 0; i < n; ++i)
   for (j = 0; j < n; ++j) data[i][j] = k;     //  заполнение матрицы
}
    //  деструктор память освобождает
Matrix::~Matrix()
{  for (int i = 0; i < n; ++i) delete[] data[i];
   delete[] data;
}
```

В реализации существуют следующие две проблемы:

1. Должна быть реализована операция умножения матрицы на вектор. Вектор может быть представлен нашим массивом `array`. Однако класс `array` не имеет доступа в закрытую часть класса `Matrix`.
2. Матрица является двумерным массивом, а операция индексирования имеет единственный аргумент: мы не можем написать `m[i, j]` — это ошибка синтаксиса.

Первая проблема решается с помощью механизма "друзей": класс `array` объявляется "другом" в классе `Matrix`:

```
friend class array;
```

Можно объявить "другом" не весь класс, а только тот метод, которому нужен доступ к закрытой части. В этом случае нужно в классе `Matrix` прописать полный заголовок метода, снабдив его словом `friend`.

Одно из решений второй проблемы — перегрузка операции `()`. Простейшая реализация выглядит так:

```
double& operator()(int i, int j) { return data[i][j]; }
double operator()(int i, int j) const { return data[i][j]; }
```

Мы реализовали две операции доступа по аналогии с операциями класса `array` — для работы с константными и не константными элементами матрицы.

Замечание

Более серьезная реализация должна, конечно, проверять правильность задания индексов.

Теперь при наличии в программе объявления

```
Matrix A(10, 12);
```

к элементам матрицы можно обращаться по индексам:

```
A(i, j) = 12;
double a = A(0, 0);
```

Можно решить проблему доступа по-другому — как советует Джеф Элджер [49]. Необходимо объявить структуру с тривиальным конструктором:

```
struct Index { int x, y; Index(int x, int y):x(x), y(y){} };
```

Тогда реализация операции индексирования будет такой:

```
double& operator[](Index i) { return data[i.x][i.y]; }
```

Операция имеет единственный аргумент, поэтому ошибок при трансляции не будет. Обращение к элементу матрицы с помощью такой операции должно быть следующим:

```
A[Index(0, 1)] = 123;
```

В качестве индекса используется анонимный временный объект, упаковавший два индекса в единственный аргумент.

Такие способы реализации конечно остроумны и демонстрируют возможности C++, но все-таки желательно иметь возможность обращаться к элементу матрицы более традиционным способом `A[i][j]`. Это можно сделать, если использовать в качестве данных наш массив `array`. В листинге 9.12 приведена реализация класса `Matrix` с массивом.

Листинг 9.12. Класс `Matrix` с массивом `array`

```
class Matrix
{  public:
   Matrix(int n = 1, int m = 1, double k = 0.0);
   ~Matrix();
   array& operator[](int i) { return data[i]; }
   private:
   array *data;
   int n;
};
Matrix::Matrix(int n, int m, double k):n(n)
{  data = new array[n];
   for (int i = 0; i < n; ++i)
   for (int j = 0; j < n; ++j) data[i][j] = k;
}
Matrix::~Matrix(){ delete[] data; }
```

Наш класс матриц определен как будто обычный одномерный массив! Тем не менее, наши матрицы являются полноценными двумерными массивами. В классе `Matrix` мы обеспечили старшую (самую левую) размерность массива, младшую размерность нам реализует класс `array`. Поэтому, несмотря на "одномерность" определения операции `[]`, мы можем обращаться к элементам матрицы "двумерным" образом:

```
int main()
{  Matrix A(10, 10);
   for (int i = 0; i < 10; ++i) A[i][i] = 1;
   for (i = 0; i < 10; ++i)
   { for (int j = 0; j < 10; ++j) cout << A[i][j] << ' '; cout << endl; }
   A[0][1] = 123;
   for (i = 0; i < 10; ++i) cout << A[0][i] << ' '; cout << endl;
   return 0;
}
```

Реализация операции индексирования позволяет нам оперировать не только отдельными элементами матрицы, но целыми строками (не столбцами!). Пусть у нас есть такие объявления:

```
array A;    //  "пустой" массив double[10]
Matrix M(10, 10);
```

И мы каким-нибудь способом заполнили матрицу и массив. Тогда можно выполнять такие присваивания:

```
A = M[i];     //  массиву A присвоена i-я строка матрицы M
M[j] = A;     //  j-я строка матрицы M равна массиву A
```

Это обеспечивается реализацией присваивания в классе `array`.

Чтобы иметь возможность использовать наш массив типа `array` в классе `Matrix`, мы должны иметь определение `array` в нашей программе. Поступим простейшим образом: используем директиву препроцессора `#include`. Назовем файл с определением класса `array` Array.cpp и скопируем его в тот же каталог, где находится файл с определением класса `Matrix`. Тогда в программе с классом `Matrix` можно написать:

```
#include;"Array.cpp"
```

В *гл. 11* мы научимся "правильно" разделять программы на файлы и "правильно" соединять их с помощью препроцессора.

Перегрузка индексирования для нецелых аргументов

Очень часто приходится решать задачи, в которых доступ к элементу набора однородных данных надо выполнять не по номеру, а по содержимому некоторого поля. Например, классической задачей является справочник телефонов. Одна запись такого справочника может иметь вид структуры:

```
struct tt { string fio; unsigned long tel; };
```

В программе есть массив таких структур:

```
tt[100000];
```

Доступ нужен по фамилии и по номеру телефона. Естественно, встроенные массивы не могут обеспечить такой доступ. Поэтому в библиотеке STL реализован контейнер `map`, в котором реализован аналогичный доступ по содержимому. Не вдаваясь в подробности реализации стандартного контейнера, реализуем простейший вариант такого механизма. Совершенно очевидно, что для этого удобнее всего перегрузить операцию индексирования. Полный класс писать не будем, покажем только схему реализации метода:

```
tt& operator[](const string &s)
{  for (int i = 0; i < 100000; ++i)
   if (tt[i].fio == s) return tt[i];
}
```

Совершенно аналогично реализуется и перегрузка операции индексирования для поля-номера телефона. Понятно, что последовательный доступ выполняется долго и медленно, но не будем разбирать работу сбалансированных деревьев, посредством которых реализован стандартный контейнер, —

для нас важно уяснить, что операцию индексирования можно перегружать и для нецелых аргументов.

Последовательный контейнер

Наш массив `array`, создание которого уже рассмотрено, хоть и динамический, но после объявления не может менять свой размер. Такая структура не годится для реализации очереди автомобилей, о которой мы говорили ранее *(см. гл. 4)*. Нам нужен последовательный контейнер, количество элементов которого меняется в процессе обработки. Элементы желательно добавлять и удалять как в начале, так и в конце контейнера. Такой контейнер называется *деком*[1] и может рассматриваться либо как очередь, либо как стек в зависимости от требуемой дисциплины доступа.

Реализуем пока минимально необходимые возможности — только операции с элементами контейнера. Нам нужны конструктор по умолчанию, деструктор, операции добавления элемента в начало и конец, операции удаления первого и последнего элемента, количество элементов, доступ к первому элементу, доступ к последнему элементу. Еще потребуется обязательная функция проверки, есть ли в контейнере элементы. Чтобы иметь возможность просмотреть весь список элементов, создадим (вспомогательную) операцию вывода на экран. Можно для удобства реализовать функцию, показывающую количество элементов в контейнере.

Как мы уже знаем, реализация такого контейнера основана на связанных списках. Создадим двусвязный список, как и в стандартной библиотеке шаблонов. Так как мы не собираемся ничего вычислять, то можно не фиксировать тип информационной части, а сделать его параметром шаблона. Простейшая структура узла очевидна:

```
struct Node
{ T item; Node *next; Node *prev; };
```

Чтобы иметь возможность инициализировать динамические переменные типа `Node`, необходимо объявить прямо в структуре простейший конструктор:

```
Node(const T &a):item(T(a)) {}
```

Информационное поле структуры инициализируется вызовом конструктора шаблонного типа в списке инициализации.

По примеру STL (STL — отличный пример для подражания) добавим в список пустой "запредельный" элемент, который будет размещаться за последним реальным элементом списка. Наличие такого элемента, на который

[1] Впервые это название использовал Дональд Кнут, и оно стало общепринятым.

постоянно ссылается указатель "хвоста", делает чрезвычайно простой проверку "пустоты" списка. Для реализации "запредельного" элемента нам требуется конструктор без аргументов. Тогда заодно реализуем и деструктор.

Учитывая возросшую сложность структуры узла, реализуем его в виде вложенного класса `Node`. Этот класс, естественно, должен быть в закрытой части класса `Deque`. Для того чтобы класс `Node` имел доступ к внутренней структуре класса `Deque`, необходимо последний прописать "другом" в классе `Node`. Таким образом, полный интерфейс нашего дека приведен в листинге 9.13.

Листинг 9.13. Интерфейс дека

```
template <class T>
class Deque
{  private:
   class Node
   {  friend class Deque;
      Node(const T &a):item(T(a)){ }
      Node(){ }
      ~Node(){ }
      T item;
      Node *next;    //  следующий элемент
      Node *prev;    //  предыдущий элемент
   };
   Deque& operator=(const Deque &);    //  запрет присваивания
   Deque(const Deque &);    //  запрет инициализации
   Node *Head;              //  "голова" списка
   Node *Tail;              //  "хвост" списка
   long count;              //  счетчик элементов
   const T dummy;           //  фиктивный элемент
   public:
   Deque():dummy(T(0))
   { Head = Tail = new Node; Tail -> next = Tail -> prev=0; count = 0; }
   Deque(const T& a)
   { Head = Tail = new Node; Tail -> next = Tail -> prev=0;
   push_front(a);
   count = 1;
   }
   ~Deque();
   bool isEmpty() const { bool t = (Head == Tail); return t; }
   long size() const { return count; }
   const T front() const
   { if (!isEmpty()) return Head -> item; else return dummy; }
   const T back() const
```

```
   { if (!isEmpty()) return Tail -> prev -> item; else return dummy; }
   void push_front(const T &a);
   void push_back(const T &a);
   void pop_front();
   void pop_back();
   void Print();
};
```

Здесь надо обратить внимание на следующие моменты:

- в приватной части класса `Deque` мы прописали константу `dummy`, которая нам нужна только для одной цели: она возвращается методами `front` и `back`, если список пуст. Необходимо заметить, что такое решение принято только потому, что мы еще не умеем обрабатывать исключения;
- мы создали оператор присваивания и конструктор копирования в приватной части класса `Deque`. Тем самым запретили присваивание одного списка другому и создание нового списка из уже существующего;
- конструкторы создают пустой "запредельный" элемент.

Замечание

Класс `Deque` транслируется без ошибок в Borland C++ Builder 6 и Visual C++ 7. Чтобы он работал в Visual C++ 6, необходимо сделать члены класса `Node` открытыми.

Реализация особых сложностей не представляет. Обратить внимание надо только на то, что в конструкторе память не выделяется. Память выделяется в операциях присоединения элемента, а возвращается в деструкторе. Поэтому в деструкторе обязательно проверять, есть ли хоть один элемент в деке:

```
template<class T>
Deque<T>::~Deque()
{  Node *delete_Node = Head;
   for (Node *p = Head; p != Tail;)
   { p = p -> next; delete delete_Node; delete_Node = p; }
   delete delete_Node;
}
```

Продемонстрируем реализацию операций с "головой" списка (их текст приведен в листинге 9.14).

Листинг 9.14. Операции с "головой" дека

```
template<class T>
void Deque<T>::push_front(const T &a)
{  Node *p = new Node(a);
```

```
   p -> next = Head;
   p -> prev = 0;
   Head -> prev = p;
   Head = p;
   ++count;
}
template<class T>
void Deque<T>::pop_front()
{  if (!isEmpty())
   {  Node *p = Head;
      Head = Head -> next;
      Head -> prev = 0;
      --count;
      delete p;
   }
}
```

Функция доступа к первому элементу прописана прямо в классе `Deque`. Реализация функций с "хвостом" дека совершенно аналогична, поэтому нет необходимости приводить их.

"Интеллектуальные" указатели

Вспомним проблемы, связанные с указателями: "потерянные" и "висячие" ссылки *(см. гл. 4)*. Как правильно указал Александреску [1], такая ситуация возникает потому, что при работе с указателями мы имеем два объекта вместо одного: сам объект и указатель на него. Указатель на динамический объект не только указывает на объект, но и *владеет* им: динамический объект создается и уничтожается только с помощью указателя. Доступ к объекту осуществляется тоже только через указатель. Следовательно, указатели на динамические переменные нельзя копировать и присваивать как попало.

К счастью, используя конструкцию класса, мы можем реализовать более "умный" указатель, чем встроенный. В этом случае мы получаем возможность управления поведением указателей:

- при создании и уничтожении;
- при копировании и присваивании;
- при разыменовании.

По сложившейся традиции такие указатели называются *"интеллектуальными"* (по англ. smart pointers).

"Интеллектуальные" указатели обычно реализуются в виде шаблона, т. к. они должны быть максимально типизированы: параметр шаблона определяет тип указываемого объекта. Пример реализации в виде шаблона приведен в листинге 9.15.

Листинг 9.15. Шаблон "интеллектуального" указателя

```
template <class T>
class SmartPtr
{  public:
   explicit SmartPtr(T* p): pointer(p){}
   SmartPtr(const SmartPtr &rhs);
   ~SmartPtr();
   SmartPtr& operator=(const SmartPtr &rhs);
   T& operator*() const
   {  // ...
    return *pointer;
   }
   T* operator->() const
   {  // ...
    return pointer;
   }
   private:
   T *pointer;
};
```

Конструктор по умолчанию отсутствует, чтобы нельзя было создать "пустой" указатель. Конструктор преобразования объявлен как `explicit`, чтобы запретить неявные преобразования встроенных типов указателей в "интеллектуальные". Конструктор копирования и операция присваивания сделаны открытыми, но при необходимости мы легко можем запретить эти действия, перенеся объявления в приватную часть класса.

Несмотря на отсутствие конструктора по умолчанию, мы все-таки можем создать нулевой "интеллектуальный" указатель, например:

```
SmartPtr<double> ip(0);
```

Тогда деструктор, текст которого приведен в листинге 9.16, обязательно должен проверять этот факт.

Листинг 9.16. Деструктор "интеллектуального" указателя

```
template<class T>
SmartPtr<T>::~SmartPtr()
{ if (!pointer) delete pointer; }
```

Однако и наши операции разыменования тоже должны учитывать возможное нулевое значение. Чтобы не завершать программу аварийно, можно объявить в приватной части фиктивный элемент-константу, который нужно

инициализировать в конструкторе. Тогда операция разыменования * может возвращать ссылку на фиктивный элемент, а операция -> будет возвращать его адрес (их текст приведен в листинге 9.17).

Листинг 9.17. Операции разыменования

```
T& operator*() const
{  if (!pointer) return *pointer;
   else return dummy;
}
T* operator->() const
{  if (!pointer) return pointer;
   else return &dummy;
}
```

Однако нам еще потребуется проверка на 0 самого "интеллектуального" указателя, чтобы в программе можно было использовать обычные конструкции вроде такой:

```
if (!ip) ...
```

Это делается довольно просто. Текст проверки приведен в листинге 9.18.

Листинг 9.18. Операция проверки на ноль

```
bool operator!() const
{ return pointer == 0; }
```

Осталось разобраться с копированием и присваиванием. Именно эти операции доставляют больше всего "головной боли" программистам. Вариантов реализации достаточно много, но сначала мы рассмотрим разрушающее копирование и присваивание (destructive copy), которое реализовано в стандартной библиотеке шаблонов STL.

auto_ptr

В STL есть шаблон "интеллектуального" указателя, который называется auto_ptr. Мейерс [24] приводит его простейшую реализацию. Нас, однако, интересует только конструктор копирования, операция присваивания и деструктор (их реализации приведены в листинге 9.19).

Листинг 9.19. Копирование и присваивание в шаблоне auto_ptr

```
template <class T>
auto_ptr<T>::auto_ptr(auto_ptr<T> &rhs)
```

```
{  pointer = rhs.pointer;        //  вступили во владение
   rhs.pointer = 0;              //  прежний владелец — больше не владелец
}
template <class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T> &rhs)
{  if (this != &rhs)
   {  delete pointer;            //  удаляем прежний объект
      pointer = rhs.pointer;     //  вступили во владение
      rhs.pointer = 0;           //  прежний владелец — больше не владелец
   }
   return *this;
}
auto_ptr<T>::~auto_ptr()
{ delete pointer; }
```

Здесь сразу бросается в глаза то, что передаваемые по ссылке параметры — не константы. Так и должно быть, чтобы можно было обнулить указатель параметра. Вторая особенность — тривиальный конструктор, который не проверяет указатель на ноль. Этого и не требуется делать, поскольку конструктор копирования и операция присваивания гарантируют "владение" объектом.

Данная реализация имеет несколько следствий. Как указывает Герб Саттер [35], следующий код будет совершенно безопасным с точки зрения утечки ресурсов:

```
void f()
{ auto_ptr<T> pt( new T); }
```

В данном случае ничего не "утекает", т. к. при выходе из функции просто вызывается деструктор `auto_ptr`, который и удаляет объект.

Однако есть и отрицательные стороны. Например, такие указатели не всегда возможно передавать в качестве параметров по значению. Так как при передаче по значению происходит копирование, то исходный указатель перестает быть владельцем объекта. Поэтому в большинстве случаев `auto_ptr` надо передавать по ссылке. Однако возвращать созданный внутри функции указатель вполне безопасно:

```
auto_ptr<T> f()
{ return auto_ptr<T> pt( new T); }
```

Таким образом, "интеллектуальный" указатель, реализованный подобным способом, не похож на обычные объекты: его копии не эквивалентны. Поэтому такие указатели нельзя использовать в качестве элементов стандартных контейнеров.

ГЛАВА 10

Исключения — что это такое

Как мы уже знаем, при выполнении любой программы бывают аварийные ситуации, вызванные самыми разнообразными причинами. Если программа используется автором, то встроить в нее обработку некоторых ошибок труда не представляет. Так мы обычно и делали, например, проверяя параметры функции. Но программы обычно пишутся не для себя, а для других. В *гл. 9* при реализации динамических классов мы неоднократно сталкивались с проблемами, когда требовалось обработать ошибочную ситуацию в некоторой функции-операции. Как правило, сама функция такую ситуацию обработать не в состоянии. Наилучшим решением было бы предоставить возможность обработать ошибочную ситуацию программе-клиенту. Необходимо только как-то сообщить ей, что возникла аварийная ситуация.

Аналогичные проблемы, как мы наблюдали, возникают и в конструкторах: если при создании объекта возникает аварийная ситуация, конструктор должен как-то сообщить об этом программе-клиенту. У конструктора, в отличие от обычных функций, нет возможности возвратить значение, сигнализирующее об аварии. Можно было бы предусмотреть в конструкторе лишний параметр специально для аварийных ситуаций, но тогда мы не сможем объявлять объекты без инициализации. Это грубое нарушение одного из важных принципов: объявления новых типов данных не должны отличаться от объявлений встроенных типов. Таким образом, и наша операция индексирования, и конструктор любого класса должны уметь сообщать об аварии каким-то другим способом, не используя для этого ни список параметров, ни возвращаемое значение.

Принципы обработки исключений

В C++ есть такие средства, которые позволяют это сделать. Эти средства образуют *механизм исключений*. В языке существуют три зарезервированных слова: `throw` (порождать), `try` (контролировать), `catch` (ловить) — которые

используются для организации обработки исключений. Общая схема такова: в одной части программы, где обнаружена аварийная ситуация, исключение генерируется; другая часть программы, которая следит за возникновением исключения, "ловит" и обрабатывает его.

Генерация исключений

Исключение — это объект. Такая точка зрения несколько расходится со здравым смыслом, т. к. обычно говорят об исключительной ситуации в программе. Однако программа генерирует объект-исключение при возникновении исключительной ситуации. Такой подход очень удобен, поскольку с объектом, в отличие от ситуации, мы можем много чего делать. Например, объявлять как обычную переменную, передать его как параметр любым из возможных способов или возвратить в качестве результата. Но, с другой стороны, для объекта-исключения мы имеем все те же проблемы, что и для обычного объекта.

Тип объекта-исключения может быть любой — как встроенный, так и определенный программистом. И это тоже очень удобно, т. к. позволяет определить собственные типы исключений, характерные именно для решаемой задачи. Генерируется объект-исключение оператором `throw`, который имеет следующий синтаксис:

```
throw выражение_генерации_исключения;
```

"Страшная" фраза `выражение_генерации_исключения` на практике означает либо константу, либо переменную некоторого типа. Например:

```
throw "Ошибка: деление на ноль!";    //  1
throw 13;                            //  2
throw s[i];                          //  3
```

В первом случае объект-исключение — это строка, которая фактически является сообщением об ошибке. Второй вариант — целая константа, которая может быть условным номером — кодом ошибки. В общем случае этот код ошибки может вычисляться, например:

```
throw 2*i*value;
```

Если все сообщения об ошибках записаны в массиве, например,

```
string s[234];
```

то в третьем случае объект-исключение тоже представляет собой строку — сообщение об ошибке. Ничто не мешает использовать в качестве объекта-исключения даже контейнер стандартной библиотеки, предварительно заполнив его необходимыми данными.

Программист может и сам определить свой собственный тип объекта-исключения, например:

```
class ZeroArgument{};
ZeroArgument ex;
if (x > 0) double y = log(x);
else throw ex;
```

Объявлять переменную необязательно, объект-исключение может быть динамическим:

```
throw new ZeroArgument();
```

Обратите внимание, что мы вызвали конструктор по умолчанию, т. к. определенный нами класс "пустой".

Замечание

Динамический объект-исключение может вызвать проблемы с возвратом памяти.

Ничто не мешает определить более развитый класс с несколькими конструкторами инициализации или копирования, и вызывать при генерации исключения один из них.

Перехват исключений

Возникновение исключения проверяется с помощью оператора `try`, с которым неразрывно связаны одна или несколько "секций-ловушек" `catch`, обычно называемые *обработчиками исключений*. Оператор `try` объявляет в любом месте программы контролируемый блок, который имеет следующий вид:

```
try { /* контролируемый блок */ }
```

После блока `try` обязательно прописывается один или несколько блоков `catch`. Форма записи "секции-ловушки" следующая:

```
catch (Спецификация_исключения) { /* блок обработки */}
```

`Спецификация_исключения` может иметь три формы, аналогичные формам передачи параметра в функцию, только параметр единственный:

```
(тип имя)     //  1
(тип)         //  2
(...)         //  3
```

Первый вариант применяется тогда, когда объект-исключение используется в блоке обработки, например, для передачи его в некоторую функцию, или для вывода информации. При этом объект-исключение может передаваться в "секцию-ловушку" любым способом: по значению, по ссылке или по указателю. Во втором варианте в блоке обработки объект-исключение никак не используется. Исключения перехватываются по типу, поэтому первые две

формы предназначены для обработки конкретного типа исключений. Если же на месте Спецификация_исключения написано многоточие (третий вариант) (как в функциях с переменным числом параметров), то такой обработчик перехватывает все исключения.

Если в блоке try во время работы не возникло исключительной ситуации, то все блоки catch пропускаются, и программа продолжает выполнение с первого оператора после всех catch. Если в блоке try генерируется исключение некоторого типа, то происходит проверка типов исключений в "ловушках". Если такой тип обнаружен, то выполняются операторы в соответствующем блоке. Если такого типа не найдено, но есть catch с многоточием, то выполняется его блок. В противном случае вызывается стандартная функция завершения terminate.

После выполнения операторов блока catch при отсутствии явных операторов перехода или оператора throw выполняются операторы, расположенные после всей конструкции try...catch. Необходимо отметить, что исключение может быть сгенерировано в одном месте программы, а обработано совершенно в другом. Блоки try...catch могут быть вложенными, причем как в блок try, так и в блок catch:

```
try {            //  блок, который может инициировать исключения
       try {     //  вложенный блок
           }
       catch(...){ }
    }
catch {           //  обработка исключения
        try {     //  вложенный блок
            }
        catch(...){ }
    }
```

Функции и исключения

Рассмотрим несколько простых примеров для демонстрации обработки исключений. Используем механизм обработки исключений для проверки параметра функции вычисления факториала. Как мы помним, набольшее целое, факториал которого можно вычислить, равно 1754. Таким образом, функция, получив число больше 1754, может сгенерировать исключение. Заодно будем проверять параметр и на нижнюю границу ноль. Текст функции для вычисления факториала с обработкой исключений приведен в листинге 10.1.

Листинг 10.1. Функция нахождения факториала с обработкой исключений

```
double f(int k)
{  try                        //  контролируемый блок
   {  if (k > 1754) throw "Аргумент > 1754!";
      if (k < 0) throw "Аргумент < 0!";
      if (k == 0) return 1;
      else return k * f(k — 1);
   }
   catch(const char *s)     //  обработчик прерываний
   { cout << ss << endl; }
}
```

Как видите, тело функции значительно усложнилось. При вызове

```
cout << f(5555) << endl;
```

на экран будет выведена строка:

```
Аргумент > 1754!
```

Это работа блока обработки исключений. Однако при создании программы в среде Visual C++ 6 после обработки исключения продолжится "нормальное" выполнение программы, и оператор вывода выведет на экран "аварийное" значение `1.#INF`. Чтобы этого не происходило, надо прописать последним оператором блока обработки оператор `throw` без выражения генерации исключения. Такой оператор можно писать только в блоке обработки исключений, иначе выполняется немедленное аварийное завершение программы. Обычно этот оператор пишется в "блоке-ловушке" тогда, когда требуется разбить обработку исключений на несколько частей. Его выполнение приводит к тому, что опять начинается поиск обработчика прерываний такого же типа, но уже вне данной функции. Если соответствующего блока `catch` не обнаружено, то программа аварийно завершается. Именно это и происходит в нашем случае. Однако мы можем и "поймать" это исключение в функции `main` и продолжить его обработку.

Таким образом, механизм исключений позволяет генерировать исключение в одном месте программы, а обрабатывать его совершенно в другом. Обычно так и поступают: функция генерирует исключение, а обработка выполняется в вызывающей функции. Текст функции для вычисления факториала с генерацией исключений приведен в листинге 10.2.

Листинг 10.2. Функция нахождения факториала с генерацией исключений

```
double f1(int k)
{  if (k > 1754) throw "Аргумент > 1754!";
```

```
   if (k < 0) throw " Аргумент < 0!";
   if (k == 0) return 1;
   else return k * f(k - 1);
}
int main()
{  try { cout << f(55555) << endl; }
   catch (const char *s)
   { cout << ss << endl; }
   return 0;
}
```

Как видим, тело функции значительно упростилось за счет существенного усложнения вызывающей программы.

Передача информации в блок обработки

Довольно часто во время обработки исключения необходимо иметь достаточно много информации. Дополнительная информация может быть чрезвычайно полезна при анализе причин сбоя программы. Например, бывает необходимо знать конкретные значения параметров функции, которые вызвали сбой программы. Однако параметр в "секцию-ловушку" передается только один. Поэтому, чтобы передать в блок обработки больше информации, мы должны реализовать собственный тип исключений в виде полноценного класса с полями и конструктором инициализации.

Рассмотрим простой пример — функцию вычисления площади треугольника по трем сторонам. Стороны треугольника — целые положительные числа, и передаются функции как параметры. Функция проверяет значения параметров и генерирует исключение с разными сообщениями об ошибках (ее текст приведен в листинге 10.3).

Листинг 10.3. Передача информации в обработчик

```
struct ErrorTriangle
{  unsigned int a, b, c;
   const char *message;
   ErrorTriangle(unsigned int x, unsigned int y, unsigned int z,
                 const char *s): a(x), b(y), c(z), message(s) {}
};
double triangle(unsigned int x, unsigned int y, unsigned int z)
{  if ((x == 0) || (y == 0) || (z == 0))
   throw new ErrorTriangle(x, y, z, "Нулевой параметр!");
   if ((x + y > z) && (x + z > y) && (y + z > x))
   {  double p = (x + y + z)/2;
```

```
      return sqrt(p * (p - x) * (p - y) * (p - z));
   }
   else throw new ErrorTriangle(x, y, z, "Не треугольник!");
}
```

Сначала мы объявили структуру с четырьмя полями. Три беззнаковых целых соответствуют сторонам треугольника, а последний параметр — для сообщения об ошибке. В структуре объявлен конструктор инициализации, который инициализирует поля. Использование этого класса продемонстрировано в теле функции. Обратите внимание на то, что оператор `throw` генерирует динамический объект-исключение, передавая ему параметры функции. Такая генерация исключения предполагает совершенно определенный способ передачи параметра в блок `catch` — по указателю. Это можно видеть в листинге 10.4.

Листинг 10.4. Передача данных в блок `catch`

```
int main()
{  try { cout << triangle(1, 1, 1) << endl; }
   catch (ErrorTriangle *e)    //  принимает динамический объект
   {  cout << e -> message << ' ';
      cout << e -> a << ';' << e -> b << ';' << e -> c << endl;
   }
   return 0;
}
```

Спецификация исключений

До сих пор мы писали заголовки функций без спецификации исключений. Если в заголовке функции не указана спецификация исключений, то функция может порождать любое исключение. Однако можно задать в заголовке явный список исключений, которые она может порождать, например:

```
void f1(void) throw(const char *s, string);
void f2(void) throw(ZeroDevide);
```

Первая функция может порождать объекты-исключения строкового типа, вторая — объект-исключение типа `ZeroDevide`.

Если в заголовке скобки спецификации исключений пустые:

```
void f1(void) throw();
```

то считается, что функция исключений не генерирует.

Если спецификация исключений объявлена в заголовке, то она должна быть указана и во всех прототипах.

К сожалению, спецификация исключений не входит в прототип функции при перегрузке, поэтому функции с различными спецификациями исключений не считаются разными. Наличие спецификации исключений, тем не менее, не является ограничением при реальной генерации исключений — функция может сгенерировать исключение любого типа. Главное, чтобы это исключение было перехвачено.

Если исключение не перехвачено, то вызывается системная функция `unexpected()`, которая вызывает `terminate()`, которая, в свою очередь, вызывает `abort()`. Мы можем подменить вызов стандартных функций. Для этого необходимо определить собственные функции, которые должны иметь заголовок:

```
void F(void)    //  имя может быть любым
```

Тогда подстановка нашей функции вместо `unexpected()` делается так:

```
set_unexpected(my_unexpected);
```

Подстановка нашей функции вместо `terminated()` практически идентична:

```
set_terminate(my_terminated);
```

После этого все не перехваченные исключения будут обрабатываться нашими функциями.

В стандартной библиотеке есть функция `uncaught_exception()`, которая возвращает `true`, если было сгенерировано исключение, которое еще не перехвачено. Ее использование особенно полезно в конструкторах и деструкторах.

Конструкторы, деструкторы и исключения

Исключения предоставляют прекрасный способ решить проблемы, связанные с ошибками во время выполнения конструкторов и при перегрузке операций. Перепишем конструкторы и операции ранее написанных нами динамических классов *(см. гл. 9)* с использованием механизма исключений. Начнем с массивов с задаваемыми границами (класс `array`). Исключительные ситуации в этом классе могли возникать в конструкторах с задаваемыми индексами, когда левый индекс был больше правого и в операции индексирования, если заданный индекс оказывался вне заданных границ. Если мы хотим различать эти случаи, то должны сначала определить классы исключений. Пусть это будут классы:

```
class bad_Index {};    //  для индекса вне границ
class bad_Range {};    //  для неправильных границ
class bad_Size {};
```

Последний тип предназначен для генерации исключений в конструкторе при задании нулевой или отрицательной длины, а так же при несовпадении

размеров массивов в операциях с двумя массивами, например, в функции вычисления скалярного произведения `product_inner`. Эти классы должны быть прописаны в публичной части класса `array`, чтобы клиент, использующий класс, мог определить перехват этих исключений. Текст новой реализации класса с генерацией исключений приведен в листинге 10.5.

Листинг 10.5. Класс `array` с генерацией исключений

```
array::array(UINT size, double k)
{  if (size > 0)
   {  left = 0;
      right = size — 1;
      size_array = size;
      data = new double[size_array];
      fill_n(data, size_array, k);
   }
   else throw new bad_Size;
}
array::array(int l, int r, double k)
{  if (l < r)
   {  size_array = (r — l) + 1;
      left = l;
      right = r;
      data = new double[size_array];
      fill_n(data, size_array, k);
   }
   else throw new bad_Range;
}
array::array(const array &a, int l, int r)
{  if ((l < r) && (a.left <= l) && (r <= a.right))
   {  left = l;
      right = r;
      size_array = (right — left) + 1;
      data = new double[size_array];
      copy(a.data + (l — a.left),
           a.data + (l — a.left) + size_array, data);
   }
   else throw new bad_Range;
}
double& array::operator[](int index)
{  if (left <= index && index <= right) return data[index];
   else throw new bad_Index;
}
```

```
double array::operator[](int index) const
{  if (left <= index && index <= right) return data[index];
   else throw new bad_Index;
}
double product_inner(const array &a, const array &b)
{  if (a.size() == b.size())
   {  double z = inner_product(a.data, a.data + a.size_array,
                               b.data, 0.0);
      return z;
   }
   else throw new array::bad_Size;
}
array operator*(const array &a, const array &b)
{  if (a.size() == b.size())
   {  array z(a.size());
      for (int i = 0; i < a.size(); ++i) z[i] = a[i] * b[i];
      return z;
   }
   else throw new array::bad_Size;
}
```

Обратите внимание, что в функциях, являющихся "друзьями" нам приходится указывать уточненное имя объекта-исключения, тогда как в функциях-методах разрешается писать имя без префикса `array`. Кроме того, нет необходимости генерировать исключения в функциях `assign()`, т. к. эту работу выполняют конструкторы.

Клиент может обрабатывать исключение `bad_Index`, например, так:

```
double a[] = { 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 };
array B(a, a + (sizeof(a) / sizeof(double)));
array G(B);
double aa;
int index = G.size();
try {  aa = product_inner(B, G);
       mm: cout << G[index] << endl;    //  1, индекс вне границ
    }
catch(array::bad_Index *e)
{  cout << "Index out of range!" << endl;
   index = 0;
   goto mm;
}
```

И здесь требуется задавать тип исключения с уточняющим префиксом `array`. Обратите внимание, что наш блок обработки выполняет не совсем

тривиальную работу. Если при выполнении оператора вывода `1` будет сгенерировано исключение типа `bad_Index` (а при первом входе в блок `try` именно так и произойдет), то на экран будет выдано сообщение об ошибке, но индексу будет присвоен `0`, и управление будет передано обратно в блок `try` для повторения выполнения оператора вывода. Теперь он сработает нормально, и будет продолжено выполнение программы, после секции `catch`.

Примечание

Система Borland C++ Builder 6 не разрешает передавать управление внутрь блока `try`, поэтому надо прописать метку вне блока.

Таким образом, мы видим, что при исключениях нет необходимости завершать программу аварийно — можно запрограммировать гораздо более гибкое поведение. Такие возможности особенно полезны при программировании реакции на исключения, в принципе не являющиеся ошибками. Например, ситуация обнаружения конца файла — это не ошибка. Или попытка получить элемент из пустой очереди — нормальная рабочая ситуация. В таких случаях программа должна, очевидно, выполнить некоторые действия (например, закрыть файл) и продолжить работу.

Оператор обработки исключения — это оператор передачи управления. Он похож на вызов функции, однако и отличается от него:

- вызов выполняется не по имени (его у блока `catch` нет), а по типу параметра;
- обработка (тело функции) может быть распределена между несколькими секциями `catch`, которые могут находиться в различных местах программы;
- так же, как и при нормальном завершении функции, возникновение исключения сопровождается вызовом деструкторов локальных объектов, этот процесс называют "раскрутка стека" (stack unwinding).

Стандартные исключения

Язык C++ предоставляет ряд стандартных исключений, которые организованы в иерархию классов. Как мы знаем, иерархическое представление часто бывает очень полезно, т. к. позволяет разбить задачу на части, повысить понимание и упростить реализацию. В любом языке программирования существуют исторически сложившиеся иерархии: вложенность блоков и вызовы функций. В C++, помимо них, можно организовать еще и иерархию классов. В этом случае используется третий механизм объектно-ориентированного программирования — наследование. Подробности наследования мы изучим в *гл. 11*, а пока рассмотрим простейший случай организации исключений.

При реализации класса `array` с генерацией исключений мы определили три типа исключений: `bad_Range`, `bad_Index` и `bad_Size`. Мы объявили их независимыми друг от друга. Но все они имеют нечто общее — эти исключения генерируются в классе `array`. Хотелось бы отразить этот факт явным образом. Это можно сделать, используя механизм наследования: мы объявим класс `array_exception`, а остальные классы — его *наследниками*:

```
class array_exception {};                          //  базовый класс
class bad_Range: public array_exception {};        //  класс-потомок
class bad_Index: public array_exception {};        //  класс-потомок
class bad_Size : public array_exception {};        //  класс-потомок
```

Чуть более сложную структуру имеет иерархия стандартных исключений, текст которой приведен в листинге 10.6.

Листинг 10.6. Иерархия стандартных исключений

```
class exception { //... };
 class logic_error : public exception { //... };
   class domain_error : public logic_error { //... };
   class invalid_argument : public logic_error { //... };
   class length_error : public logic_error { //... };
   class out_of_range : public logic_error { //... };
 class runtime_error : public exception { //... };
   class range_error : public runtime_error { //... };
   class overflow_error : public runtime_error { //...}
   class underflow_error : public runtime_error { //... };
 class bad_cast : public exception { //... };
 class bad_alloc : public exception { //... };
 class bad_tipeid : public exception { //... };
 class bad_exception : public exception { //... };
 class iss_base::failure : public exception { //... };
```

Эта иерархия служит основой для создания собственных исключений и иерархий исключений. Мы можем определять свои собственные исключения, унаследовав их от класса `exception`. Для этого в программе надо прописать такой оператор:

```
#include <stdexcept>
```

Вспомним, что с блоком `try` может быть связано несколько "секций-ловушек" `catch`. При наличии иерархии исключений мы должны писать блоки `catch` в правильном порядке: сначала (ближе к блоку `try`) прописываются более специализированные обработчики, затем — более общие. Последним может стоять универсальный блок `catch` с многоточием.

Из всех стандартных исключений наиболее часто используются следующие:

- исключение `bad_alloc`, генерируемое операцией `new` при невозможности выделить требуемую память;
- исключения `bad_cast` и `bad_typeid`, генерируемые механизмами RTTI (Run-Time Type Informatiom — *см. гл. 11*);
- исключения группы `runtime_error`, которые программа-клиент может генерировать при наступлении соответствующей исключительной ситуации.

Стандартные исключения включают функцию-метод `what()`, использование которой продемонстрируем на примере из справочника интегрированной среды Borland C++ Builder 6. Текст примера приведен в листинге 10.7.

Листинг 10.7. Использование метода `what()`

```
#include <stdexcept>
#include <iostream>
#include <string>
using namespace std;
void f() { throw runtime_error("a runtime error"); }
int main ()
{  string s;
   try { s.replace(100, 1, 1, 'c'); }
   catch (const exception& e)
   { cout << "Got an exception: " << e.what() << endl; }
   try { f(); }
   catch (const exception& e)
   { cout << "Got an exception: " << e.what() << endl; }
   return 0;
}
```

Эта программа выведет на экран две строки:

```
Got an exception: basic_string
Got an exception: a runtime error
```

Первая строка — это результат работы первого обработчика при неправильном использовании метода `replace` в строке `s`, вторая — это уже результат явной генерации исключения типа `runtime_error` в функции `f`. Обратите внимание на одну замечательную особенность блоков `catch`: в качестве типа формального параметра используется базовый класс, тогда как при выполнении в обработчик попадает объект-исключение совсем другого типа — наследника от базового. Это очень важное свойство наследования мы рассмотрим в *гл. 11*.

ГЛАВА 11

Наследование

В предыдущих главах мы довольно часто употребляли два слова: инкапсуляция и полиморфизм. Мы писали наши программы, следуя принципу инкапсуляции, который в простейшем виде требует сокрытия информации. В C++ реализовано несколько языковых механизмов, позволяющих управлять инкапсуляцией. Механизм перегрузки функций и операций воплощает реализацию полиморфизма. Третьим важнейшим механизмом C++ является *наследование*. Наследование в любом современном языке программирования выполняет две роли: с одной стороны, предотвращает дублирование кодов, а с другой — позволяет развивать работу в нужном направлении.

При наследовании обязательно имеется *класс-предок* (родитель) и *класс-наследник* (потомок). Класс-предок часто называют еще "суперклассом" (или порождающим классом), а класс-наследник — подклассом (порожденным классом). В C++ принято порождающий класс называть *базовым*, а порожденный класс — *производным*. Мы будем использовать все эти названия по мере необходимости.

Отношения между родительским классом и его потомками называются *иерархией наследования*. Мы уже познакомились с простейшим вариантом этого механизма, когда рассматривали иерархию исключений. Вообще говоря, глубина наследования ничем не ограничена: мы можем наследовать столько раз, сколько требуется для решения нашей задачи.

Простое наследование

Простым называется наследование, при котором производный класс имеет только одного родителя. Формально наследование одно класса от другого можно задать следующей конструкцией:

```
class имя_класса_потомка: [модификатор_доступа] имя_базового_класса
{ тело_класса }
```

Класс-потомок наследует структуру (все элементы данных) и поведение (все функции-методы) базового класса. *Модификатор доступа* определяет доступность элементов базового класса в классе-наследнике. Квадратные скобки говорят о том, что этот модификатор может отсутствовать. Этот модификатор мы будем называть *модификатором наследования.*

Мы уже знакомы с двумя модификаторами доступа: `public` (общий) и `private` (личный). При наследовании используется еще один — `protected` (защищенный). Как и в реальной жизни, некоторые из свойств являются общими для всех объектов (данного класса), другие — специфическими, присущими только конкретному объекту. Третий вид свойств присущ некоторому подмножеству объектов. Типичным аналогом является семья, состоящая из нескольких поколений: с одной стороны, все члены семьи имеют общую фамилию и участвуют в общественной жизни (`public`). С другой стороны, у каждого из членов семьи есть свои личные (`private`) интимные тайны. Однако члены семьи являются родственниками и участвуют еще и в общей семейной жизни, которая, однако, не является доступной обществу. Именно "семейная жизнь" объектов потребовала наличия модификатора `protected`.

У нас есть четыре варианта наследования: класс от класса, класс от структуры, структура от структуры и структура от класса. В зависимости от модификаторов доступа в объявлении базового класса и при наследовании, доступность элементов базового класса из классов-наследников изменяется. В табл. 11.1 приведены все варианты доступности элементов базового класса в производном классе.

Таблица 11.1. *Доступ к элементам базового класса в классах-наследниках*

Модификатор в базовом классе	Модификатор наследования	Доступ в производном классе	
		`struct`	`class`
`public`	Отсутствует	`public`	`private`
`protected`	Отсутствует	`public`	`private`
`private`	Отсутствует	Недоступны	Недоступны
`public`	`public`	`public`	`public`
`protected`	`public`	`protected`	`protected`
`private`	`public`	Недоступны	Недоступны
`public`	`protected`	`protected`	`protected`
`protected`	`protected`	`protected`	`protected`
`private`	`protected`	Недоступны	Недоступны

Таблица 11.1 *(окончание)*

Модификатор в базовом классе	Модификатор наследования	Доступ в производном классе	
		`struct`	`class`
`public`	`private`	`private`	`private`
`protected`	`private`	`private`	`private`
`private`	`private`	Недоступны	Недоступны

Общепринятой практикой, однако, является правило — не наследовать от структур, а только от классов. Мы тоже будем этому правилу следовать. Если в качестве модификатора доступа записано слово `public`, то такое наследование называется *открытым*. Соответственно, при использовании модификатора `protected` имеем *защищенное наследование*, а слово `private` определяет *закрытое наследование*.

Открытое наследование

Наследование устанавливает некоторое отношение между базовым классом и наследником. В практике объектно-ориентированного программирования в подавляющем большинстве случаев используется открытое наследование. Такое наследование устанавливает между классами отношение "является": класс-потомок "является" разновидностью класса-родителя, но не наоборот. С. Мейерс [23] пишет, что "единственным наиболее важным правилом в объектно-ориентированном программировании на C++ является следующее: открытое наследование означает "класс есть разновидность класса". Когда мы пишем, что класс `Derived` (производный) открыто наследует от класса `Base` (базовый), мы сообщаем компилятору, что каждый объект типа `Derived` является также объектом класса `Base`. Обратное — неверно! Например, спортсмен является человеком, но не всякий человек — спортсмен. Аналогично реактивный самолет все-таки является самолетом, но не всякий самолет — реактивный. Когда мы работаем в системе Windows, то часто имеем дело с диалоговыми окнами, но не всякое окно является диалоговым.

Таким образом, везде, где может быть использован объект типа `Base`, может применяться и объект типа `Derived`. На простом примере, приведенном в листинге 11.1, это хорошо видно.

Листинг 11.1. Демонстрация отношения "является"

```
class Clock {...};                    //  класс часов
class Alarm: public Clock {...};      //  класс будильников
void settime(Clock &c, time t);       //  установка времени часов
```

```
void setalarm(Alarm &a, time t);    //  установка будильника
Clock C;                            //  объект-часы
Alarm A;                            //  объект-будильник
settime(C, t);                      //  ставим время часов
settime(A, t);                      //  ставим время на часах-будильнике
setalarm(A, t);                     //  установка стрелки будильника
setalarm(C, t);                     //  ошибка: часы — не будильник
```

Так как не всякие часы являются будильником, то в последней строке компилятор выдаст ошибку.

В простых вариантах открытое наследование предполагает некоторые изменения в производном классе по сравнению с базовым классом: обычно добавляют новые данные, изменяют или добавляют методы. Простейший класс `Base` и два наследника от него представлены в листинге 11.2.

Листинг 11.2. Варианты открытого наследования

```
class Base
{  public:
   Base():a(-1){}
   void Print() { cout << a << ';'; }
   private: int a;
};
class D01: Base
{  public:
   void Print(){ cout << b << endl; }
   void set(float b) { this -> b = b; }
   private: float b;
};
class D02: Base
{  public:
   void Print() { Base::Print(); cout << a << endl; }
   void set(int a) { this -> a = a; }
   private: int a;
};
```

В базовом классе определен простейший конструктор по умолчанию и метод `Print()`, позволяющий выводить поле данных на экран. Определены два наследника базового класса: `D01` и `D02`. В `D01` добавлено поле данных `float b` и переопределен метод `Print()`. Кроме того, добавлен новый метод `set()`.

В классе-наследнике `D02` мы видим более интересные изменения: новое поле может иметь такое же имя и даже быть такого же типа, что и поле в базовом классе. Естественно, аналогично локальным переменным новое поле

в классе-наследнике скрывает поле родительского класса. Если в дочернем классе все-таки необходимо обращаться к одноименному полю родительского класса, то надо использовать операцию разрешения контекста ::. Эта же операция используется и для доступа к одноименным методам родительского класса. В D02 переопределен метод Print(), в котором выполняется вызов одноименного метода родительского класса. Вызов без квалификатора

```
void Print()
{ Print(); cout << a << endl; }
```

является рекурсивным и приведет к аварийной остановке программы. Метод Print() класса D02 выведет на экран все поля данных: и унаследованные от базового класса, и определенные в наследнике. Метод Print() класса D01 выведет на экран только поля D01.

Конструкторы и деструкторы в производных классах

Конструкторы, деструкторы и операция присваивания классов, участвующих в наследовании, представляют особый случай — они не наследуются. Начнем с конструкторов. Если в базовом классе нет конструкторов или есть только конструктор без аргументов, то в производном классе конструктор можно не писать. Именно такой случай приведен в листинге 11.2. Система в этом случае создает в классе-наследнике простейший конструктор по умолчанию и в нем вызывает соответствующий конструктор базового класса.

Второе: если в базовом классе все конструкторы — с аргументами, то производный класс обязан иметь конструктор, в котором явно должен быть вызван конструктор базового класса. Это может быть сделано либо в теле конструктора-наследника, либо в его списке инициализации. Давайте рассмотрим простую программу, текст которой приведен в листинге 11.3.

Листинг 11.3. Конструкторы в производном классе

```
class Point2D
{  public:
    //  Point2D(): x(0.0), y(0.0){}    //  A
   Point2D(double x, double y): x(x), y(y) {}
   Point2D(const Point2D &t): x(t.x), y(t.y) {}
   void Print() const { cout << '<' << x << ',' << y << '>'; }
   double getx() const { return x; }
   double gety() const { return y; }
   private:
   double x, y;
};
```

```
class Point3D: public Point2D
{  public:
   Point3D(double x, double y, double z): Point2D(x, y), z(z){}
   double getz() const { return z; }
   private:
   double z;
};
```

Мы определили простой класс "Точка на плоскости" с именем `Point2D`, в котором реализованы два конструктора: конструктор инициализации и конструктор копирования. В производном классе "Точка в пространстве" с именем `Point3D` реализован конструктор инициализации, в котором конструктор базового класса вызывается явным образом в списке инициализации. Не будет ошибки вызвать конструктор базового класса и в теле конструктора-наследника:

```
Point3D(double x, double y, double z): z(z) { Point2D(x, y); }
```

Необходимость явно вызывать конструктор базового класса обусловлена тем, что поля базового класса нельзя инициализировать в списке инициализации класса-наследника. В данном случае, т. к. поля закрыты, их нельзя явно инициализировать и в теле конструктора. Для защищенных (`protected`) полей инициализация в теле конструктора класса-наследника допустима, например:

```
Point3D(double x, double y, double z):z(z)
{ this -> x = x; this -> y = y; }
```

Обратите внимание, что ни в базовом классе, ни в классе-наследнике не определен конструктор по умолчанию. Это приводит к тому, что объявления

```
Point2D a;
Point3D b;
```

сопровождаются сообщениями об ошибках трансляции. Если мы раскомментируем конструктор по умолчанию в базовом классе (строка `A`), то создавать объект базового класса без инициализации будет можно, а объект производного класса — нельзя:

```
Point2D a;    //  работает
Point3D b;    //  по-прежнему не работает
```

Аналогичная картина наблюдается и с операцией присваивания. Эта операция не определена в нашем примере ни в базовом, ни в производном классе. Определим два объекта и попробуем присвоить их друг другу:

```
Point2D b(1, 2);
Point3D d(3, 4, 5);
```

```
b = d;    //  работает
d = b;    //  не работает
```

Первое присваивание срабатывает, поскольку для базового класса система создала операцию по умолчанию. Но при выполнении этой операции происходит так называемая "срезка": присваиваются только те поля, которые определены в базовом классе.

Почему не работает второе присваивание, поначалу не совсем ясно. Вроде бы производный класс "шире", чем базовый, и поля объекта базового класса вполне могли бы быть присвоены соответствующим полям объекта производного класса. Ситуация становится очевидной, если написать второе присваивание в функциональной форме:

```
d.operator=(b);
```

Так присваивание не наследуется, и мы не определили операцию в классе-наследнике явным образом, данное присваивание означает вызов функции, которая не определена. Естественно, компилятор выдаст соответствующее сообщение об отсутствии определения.

Несмотря на то, что мы не определяли в производном классе методов `getx()` и `gety()`, мы вполне можем ими пользоваться (по наследству):

```
Point3D d(1, 2, 3);
cout << d.getx() << ',' << d.gety() << ',' << d.getz()endl;
```

На экран будет выведено:

```
1, 2, 3
```

Осталось разобраться с деструкторами, а также порядком вызовов конструкторов и деструкторов. Во-первых, деструкторы, как и конструкторы, не наследуются; однако при отсутствии определения система сама формирует деструктор по умолчанию. Во-вторых, даже если мы определим собственный деструктор, явно вызывать деструктор базового класса нет необходимости — это делается автоматически. В-третьих, порядок вызова конструкторов при создании объекта строго определен: сначала конструктор базового класса, затем порожденного. Уничтожение объектов (вызов деструкторов) выполняется в обратном порядке. Это легко видеть на простом примере, текст которого приведен в листинге 11.4.

Листинг 11.4. Порядок вызова конструкторов и деструкторов

```
class Base
{  public: Base() { cout << "Base()" << endl; }
   ~Base() { cout << "~Base()" << endl; }
}
```

```
class Derived: public Base
{  public: Derived() { cout << "Derived()" << endl; }
   ~Derived() { cout << "~Derived()" << endl; }
}
int main()
{  derived a;
   return 0;
}
```

Эта программа выведет на экран:

```
Base()
Derived()
~Derived()
~Base()
```

Как видите, "рождение" и "смерть" объектов происходят по принципу LIFO: "последним создан — первым уничтожен".

Закрытое наследование

Ранее было сказано, что открытое наследование C++ рассматривается как отношение типа "класс есть разновидность класса". В частности, объект производного класса может быть неявно преобразован в объект базового класса, например, при передаче параметров (см. листинг 11.1). Заменим в этом примере открытое наследование на закрытое и посмотрим, к чему это приведет:

```
class Clock {...};                  //  класс часов
class Alarm: private Clock {...};   //  класс будильников
void settime(Clock &c, time t);     //  установка времени часов
Clock C;                            //  объект-часы
Alarm A;                            //  объект-будильник
settime(C, t);                      //  ставим время часов
settime(A, t);                      //  ставим время на часах-будильнике
```

Неожиданно оказывается, что объект типа `Alarm` не является объектом типа `Clock`. Во всяком случае, автоматическое преобразование не производится. Можно это преобразование выполнить явно,

```
settime(reinterpret_cast<Clock &>(A), t);  //  ставим время на будильнике
```

но нас интересует не преобразование, а отношение между классами, которое реализует закрытое наследование.

Примечание

Преобразование `static_cast` не "проходит" — это подтверждает тот факт, что объекты типа `Alarm` и объекты типа `Clock` не являются "родственниками".

Если класс D является закрытым наследником класса B, то это означает, что объекты типа D реализуются посредством объектов типа B. Обычно отношение "класс реализован посредством класса" моделируется с помощью вложения. Например, карточка сотрудника в отделе кадров обычно содержит имя, фамилию и отчество, дату рождения, адрес проживания и т. д. При реализации мы можем выразить эти данные следующим образом:

```
class Date { ... };        //  даты
class Address { ... };     //  адрес места жительства
class Person {
public: ...
private:
string name;             //  вложенный объект
Date birthday;           //  вложенный объект
Address address;         //  вложенный объект
             };
```

В этом случае говорят, что класс `Person` реализован посредством вложения классов `Date`, `Addres` и `string`. Не путайте с языковым механизмом вложенных классов (см. листинг 9.11). В качестве синонимов термина "вложение" часто используют слова "композиция" и "включение".

Однако в некоторых случаях без наследования не обойтись, поэтому давайте разберемся, что нам дает закрытое наследование, кроме того, что "класс реализован посредством класса"? В *гл. 9* мы описали реализацию дека в виде шаблона. Шаблоны — один из наиболее полезных инструментов C++, недаром библиотека STL реализована именно этим способом. Однако активное использование шаблонов часто приводит к "раздуванию" кода, т. к. на пять случаев инстанцирования одного шаблона мы получаем пять практически идентичных копий его кода. Как пишет Мейерс [23], избавиться от многократного дублирования кода можно, если использовать бестиповые указатели и закрытое наследование. При этом:

- создается один класс, который хранит бестиповые указатели;
- создается производный класс-шаблон посредством закрытого наследования, чтобы обеспечить строгий контроль типов.

Однако где же здесь экономия, если там шаблон, и здесь шаблон? Рассмотрим пример Мейерса, приведенный в листинге 11.5.

Листинг 11.5. Закрытое наследование

```
class GenegicStack
{  protected:    //  конструктор "в открытую" вызывать нельзя
   GenericStack();
   ~GenericStack();
```

```
   void push (void * obj);      //  в стек
   void * pop();                //  из стека
   bool empty() const;          //  стек пуст?
   private:
   struct Node {                //  узел списка
   void * data;                 //  данные
   Node * next;                 //  связь со следующим узлом
   Node (void * Data, Node * Next)
         :data(Data), next(Next) {}    //  конструктор узла
};
   Node * top;    //  "голова" списка — вершина стека
    //  запрет копирования и присваивания
   GenericStack(const GenericStack &rhs);
   GenericStack7 operator=( const GenericStack &rhs);
};
template <class T>
class Stack: private GenericStack
{  public:
   void push (T * obj) { GenericStack ::push(object); }    //  в стек
   T * pop()
   { return static_cast<T*>(GenericStack::pop()); }        //  из стека
   bool empty() const { return GenericStack::empty(); }    //  стек пуст?
};
```

В данном случае мы имеем единственный класс-реализацию (`GenericStack`) и множество классов-интерфейсов, получающихся при инстанцировании шаблона. Размножаются только вызовы методов класса-реализации, что является приемлемой платой за безопасность типов. С другой стороны, этот код максимально поддерживает принцип инкапсуляции: реализация клиенту недоступна, и разработчик может менять ее по своему усмотрению — клиентский код от этого не изменяется. Как указывает тот же Мейерс [23], "код разработан таким способом, что он является максимально эффективным, а также поддерживает строгий контроль типов".

Легко модифицировать данный пример для нашего дека.

Виртуальность

В C++ без указателей обойтись чрезвычайно трудно. Как мы только что в очередной раз убедились (и убедимся еще не раз), применение указателей позволяет написать более универсальный код, существенно повышает уровень инкапсуляции, способствует повторному использованию кода. При открытом наследовании указатели играют важнейшую роль. Предположим, что у нас в программе, текст которой приведен в листинге 11.6, определен класс `Base` с методом `Print()` и класс-наследник `Derived`, в котором этот метод переопределен.

Листинг 11.6. Доступ к методу наследника

```
class Base
{  public:
   Base(int x): a(x) {}
   void Print()const { cout << a <<';'; }
   private:
   int a;
};
class Derived: public Base
{  public:
   Derived (int x, float y): Base(x), b(y) {}
   void Print()const { cout << b << ';'; }
   private:
   float b;
};
```

Определим объекты обоих типов и указатели на них:

```
Base ob(1), *pb;          //  объект и указатель базового типа
Derived(2, 2.0), *pd;     //  объект и указатель производного типа
```

Присваивание для указателей работает так же, как и для объектов: указателю производного типа можно присваивать адреса объектов только производного типа, а вот указатель базового типа может хранить адреса объектов как базового, так и производного типа. Это, например, позволяет иметь контейнер указателей на базовый тип, а во время работы программы помещать туда указатели на любой производный тип. Тем самым мы можем "забыть" о потенциально опасных бестиповых указателях, используя вместо них указатели на базовый тип иерархии.

Пока все хорошо, но попытаемся выполнить небольшой фрагмент программы:

```
pb = new Base(2);              //  базовый -> базовый
pb -> Print();                 //  Base::Print()
pb = new Derived(3, 3.0);      //  базовый -> производный
pb -> Print();                 //  опять Base::Print()
```

Выясняется, что вызов метода происходит всегда по типу указателя, а не по типу присваиваемого адреса (объекта). Чтобы лучше понять проблему, обратимся к традиционному примеру: геометрическим фигурам. Есть базовый класс `Shape` (форма) и наследники от него — `Ellipse` (элипс) и `Rectangle` (прямоугольник). Функции рисования эллипса и прямоугольника, естественно, должны отличаться, поэтому они обязательно переопределяются в производных классах.

Чтобы вызвать метод производного класса через указатель базового, требуется выполнить явное преобразование типа указателя:

```
((Derived *)pb) -> Print();
```

или

```
static_cast<Derived *>(pb) -> Print();
```

Примечание

Оператор `static_cast` работает, т. к. объекты и указатели — "родственники", принадлежащие одной иерархии классов.

Однако это не всегда возможно, т. к. во время выполнения программы указатель может ссылаться на объекты разных классов иерархии, и во время компиляции конкретный класс неизвестен.

Еще одно традиционное решение заключается в использовании перечислимого типа и оператора-переключателя. Схема этого решения приведена в листинге 11.7.

Листинг 11.7. Поле типа для реализации принципа полиморфизма

```
class Base
{  enum Object_type { B, D };     //  определение перечислимого типа
   public:
   Base():type(B) {}              //  тип объекта — базовый
    //...
   private: //...
   Object_type type;              //  текущий тип объекта
};
class Derived: public Base
{  public:
   Derived(){ type = D; }         //  тип объекта — производный
    //...
};
```

Тогда схема реализации функции вывода будет такой:

```
void Print(const Base *po)
{  switch(po -> type)
   {  case Base::D:    //  вывод полей производного класса
      case Base::B:    //  вывод полей базового класса
      break;
   }
}
```

Мы специально не поставили `break` после вывода полей производного класса, т. к. обычно требуется сразу вывести поля базового класса.

Функция является глобальной и обрабатывает объекты всех типов. Это прекрасно работает для небольшой иерархии классов в программе, которую сопровождает один человек. Однако в большой иерархии могут быть реализованы сотни функций, работа которых зависит от типа аргумента. Если потребуется изменить иерархию классов добавлением нового класса-наследника, поиск и модификация таких функций превращаются в большую проблему.

Для решения проблемы автоматического вызова нужного метода через указатель на объект базового класса, в язык C++ были добавлены так называемые *виртуальные* функции-методы. Для того чтобы сделать функцию виртуальной, достаточно прописать спецификатор `virtual`, который является зарезервированным словом. Добавим этот спецификатор к определению функций `Print()` в примере, текст которого приведен в листинге 11.6:

```
virtual void Print() const    //  в базовом классе
{ cout << a << ';'; }
virtual void Print() const    //  в производном классе
{ Base::Print(); cout << b << ';'; }
```

В производном классе необязательно писать слово `virtual`, но его написание — это хороший стиль программирования. Тогда следующий фрагмент

```
pb = new Base(2);             //  базовый -> базовый
pb -> Print();                //  Base::Print()
pb = new Derived(3, 3.0);     //  базовый -> производный
pb -> Print();                //  Derived::Print()
```

сработает именно так, как и предполагалось с самого начала: первый вызов — это вызов метода базового класса, второй — вызов метода производного класса.

Осталась одна небольшая проблема: мы должны возвратить память системе:

```
delete pb;
```

Однако при этом возникает точно такая же ситуация, как и при вызове метода производного класса через указатель базового класса: должен быть вызван деструктор производного класса по указателю базового класса. В обоих случаях решение одинаковое — нужно объявить в базовом классе *виртуальный деструктор*:

```
virtual ~Base(){};
```

Без такого объявления уничтожение динамических объектов может вызвать проблемы, наименьшая из которых — нарушение целостности и согласованности динамической памяти.

Правила описания и использования виртуальных методов следующие.

1. Виртуальная функция может быть только методом класса.
2. Любую перегружаемую операцию-метод класса можно сделать виртуальной.
3. Виртуальная функция наследуется; значит, замещать ее не обязательно.
4. Если в базовом классе определена виртуальная функция, то метод производного класса с таким же именем и прототипом (включая и тип возвращаемого значения) автоматически является виртуальным (слово `virtual` указывать необязательно) и замещает функцию-метод базового класса.
5. Конструкторы не могут быть виртуальными.
6. Деструкторы могут (чаще — должны) быть виртуальными — это гарантирует корректный возврат памяти через указатель базового класса.

Таким образом, виртуальные функции — это еще одно проявление полиморфизма в C++. Александреску [1] указывает, что в C++ реализованы два типа полиморфизма:

- *статический полиморфизм* (compile-time polymorphism — полиморфизм времени компиляции), который осуществляется с помощью перегрузки и шаблонов функций;
- *динамический полиморфизм* (run-time polymorphism — полиморфизм времени выполнения), реализуемый виртуальными функциями.

Класс, включающий виртуальные функции, называется *полиморфным классом*.

RTTI и *dynamic_cast*

Использование виртуальных функций решает одни проблемы, но порождает другие: теперь нам неизвестен тип объекта, на который ссылается указатель. Однако иногда требуется во время выполнения программы либо уметь преобразовать этот тип к нужному, либо просто проверить, тот ли тип у объекта, на который ссылается указатель. C++ позволяет делать и то, и другое. Использование информации о типе во время выполнения программы обычно называют английской аббревиатурой RTTI (Run-Time Type Information — информация о типе на этапе выполнения).

Преобразование к нужному типу выполняется оператором[1] динамического преобразования `dynamic_cast`, который можно применять к указателям и ссылкам на объекты родственных полиморфных классов. Преобразование

[1] Мы используем термин "оператор" по примеру книги [37].

указателей, не являющихся "родственниками", не допускается. Преобразование указателей на не полиморфные классы (не включающих виртуальных функций) тоже не допускается — для этого предназначены операторы `static_cast` и `reinterpret_cast`. Следовательно, оператор нельзя применять для преобразования встроенных типов. Формат оператора для преобразования указателей следующий:

```
dynamic_cast<тип*>(указатель)
```

Указатель в круглых скобках преобразуется к типу в угловых скобках. Этот тип должен быть родственным для указателя. Преобразование из базового класса в производный называют *понижающим*, приведение из производного к базовому классу — *повышающим*, а приведение между производными классами одного базового — *перекрестным*. Если преобразование не удается (или указатель равен нулю), то оператор выдает ноль. Следовательно, результат преобразования надо всегда проверять.

Б. Страуструп утверждает [37], что приведение `dynamic_cast<T*>(p)` для указателей можно интерпретировать как вопрос: "Объект, на который указывает `p`, имеет тип `T`?" Аналогичное приведение для ссылок `dynamic_cast<T&>(r)`, по словам Б. Страуструпа, является не вопросом, а утверждением: "Объект, на который ссылается `r`, имеет тип `T`." Это означает, что результат приведения проверяется самой операцией, и если что не так — генерируется исключение. А "ловить" исключения мы уже умеем.

Другим средством C++, позволяющим получать и обрабатывать информацию о типе на этапе выполнения, является операция `typeid()`. Чтобы ее использовать, необходимо включить заголовок `<typeinfo>`. В этой библиотеке определен класс `type_info`, который предоставляет ряд простых операций для манипулирования информацией о типах на этапе выполнения. Использование операций этого класса и операции `typeid()` демонстрирует пример, приведенный в справочной системе интегрированной среды Borland C++ Builder 6. Немного модифицированная версия программы приведена в листинге 11.8

Листинг 11.8. Использование операции `typeid`

```
#include <iostream>
#include <typeinfoюр>
#include <conio.h>    //  для остановки по getch()
using std::cout;
using std::endl;
class A               //  базовый класс
{ virtual void f(void){}; };
class B: public A     //  класс-наследник
{ virtual void f(void){}; };
```

```
void main() {
char C;
float X;
  //  использование type_info::operator==() для сравнения типов
if (typeid(C) == typeid(X))
cout << "C и X одного типа." << endl;
else cout << "C и X НЕ одного типа." << endl;
  // сравнение имен типов: "double" < "int"
cout << " Перед " << typeid(A).name() << ":" <<
        (typeid(A).before(typeid(A)) ? true : false) << endl;
A a, *pA;
B b, *pB;
pA = &a;
cout << typeid(*pA).name();    //  выводит A
pB = &b;
cout << typeid(*pB).name();    //  выводит B
pA = &b;
cout << typeid(*pA).name();    //  выводит B
getch();    //  остановка для просмотра результатов
}
```

В примере определены два полиморфных класса: базовый `A` и производный `B`. Программа сначала сравнивает типы переменных встроенных типов и сообщает, что эти типы не совпадают. Затем лексикографически сравниваются имена типов `A` и `B` и, естественно, выводит, что `A` перед `B` — это истина. После этого объявляются объекты и указатели классов `A` и `B`. Указателям присваиваются адреса объектов и имена типов выводятся на экран. Самый важный случай — последний, в котором указателю на базовый класс присваивается адрес объекта производного типа. Оператор вывода совершенно правильно выводит `B`.

Большие программы

Как мы знаем, невозможно написать действительно большую программу, не разбивая ее на части. *Механизмы декомпозиции* бывают двух видов: *логические* и *физические*. Функции и классы — это два известных механизма логической декомпозиции. Однако большие программы требуется разбивать на части не только логически, но и физически. Дело не только в объеме (миллион строк по 60 символов составляют 60 Мбайт, что при современном развитии техники не является проблемой). Более критическим ресурсом является время. Вы можете себе представить, сколько часов потребуется компилятору, чтобы оттранслировать программу в миллион строк? Поэтому разделение программы на части — это действительно необходимость.

Отдельная часть большой и сложной программы называется *единицей трансляции*. В C++ нет никаких специальных конструкций для обозначения единицы трансляции — таковой считается файл. Таким образом, мы можем разбить большую программу на файлы и транслировать их по отдельности. Такой способ называется *раздельной трансляцией*. Но отдельно транслируемые модули сами по себе работать не будут (хотя бы потому, что главная функция только одна и может размещаться только в одном файле) — программу требуется скомпоновать (собрать). Обычно эту работу выполняет специальная программа-компоновщик, которая обязательно входит в состав любой интегрированной среды.

Примечание

На программистском жаргоне российских программистов компоновщик называют *линкером* — от английского linker (связыватель).

При компоновке в программу собираются не только наши модули, но и стандартные. Если вы напишете в программе вызов функции `sin(x)`, то компоновщик разыщет среди стандартных модулей тот, в котором записана эта функция, и присоединит ее.

Организацию программы в виде набора исходных файлов обычно называют *физической структурой программы*. Физическое разбиение программы на файлы обычно определяется ее логической структурой, однако так бывает не всегда — все определяется конкретной ситуацией.

Раздельная компиляция требует согласования объявлений. Например, в одном из модулей может быть определена некоторая функция `f`. Тогда в других модулях можно объявлять прототипы этой функции, причем все они должны в точности соответствовать заголовку определения. Во всех единицах трансляции должны быть согласованы объявления не только функций, но и переменных, классов, шаблонов, перечислений и пространств имен.

Переменные, функции и файлы

Все функции по умолчанию являются глобальными (всем именам функций по умолчанию присваивается класс памяти `extern`). Имена функций видны во всех точках программы и, как имена внешние, должны быть уникальными в программе.

Функциям можно присваивать атрибут `static`. Как известно, в C++ модулем является файл. Пока программа состоит из одного модуля-файла, атрибут `static` не играет никакой роли. Но если программа включает несколько модулей, то этот атрибут ограничивает видимость имени функции только тем модулем, где она определена. Таким образом, атрибут `static` по своему действию похож на `private` (атрибут доступа для элементов класса). Пусть, например, программа состоит из двух модулей — файлов m1.cpp и m2.cpp

(не указаны прототипы, которые, конечно же, должны быть прописаны в модулях):

```
  //  модуль m1.cpp
void f1(void) {...};          //  определение глобальной функции
static void f2(void){...};    //  определение локальной функции
  //  модуль m2.cpp
f1();                         //  вызов глобальной функции
f2();                         //  ошибка — вызов невидимой функции
```

Имя функции с атрибутом `static` должно быть уникальным в данном модуле, но может повторяться в других модулях. Более того, аналогично принципу локализации переменных, локальная в модуле функция (с атрибутом `static`) перекрывает глобальную в пределах модуля (аналогично тому, как локальная переменная в теле функции перекрывает глобальную). Например, в следующем примере функция `f1` в модуле m2.cpp перекрывает глобальную функцию, определенную в модуле m1.cpp:

```
  //  модуль m1.cpp
void f1(void) {...};          //  определение глобальной функции
static void f2(void){...};    //  определение локальной функции
f1();                         //  вызов глобальной функции
  //  модуль m2.cpp
static void f1(void) {...};   //  определение локальной функции
f1();                         //  вызов локальной функции
f2();                         //  ошибка — глобальной функции нет
  //  модуль m3.cpp
static void f2(void) {...};   //  определение локальной функции
f1();                         //  вызов глобальной функции
f2();                         //  вызов локальной функции
```

Естественно, возникает вопрос: если атрибут `static` делает имя функции локальным в модуле, то нельзя ли сделать локальным имя главной функции `main`. Вопрос не "праздный": в языке Java в каждом классе можно иметь собственную функцию `main`, что очень полезно с точки зрения независимой отладки функций модуля. К сожалению, ни одна интегрированная среда не позволяет проделать "в лоб" такой фокус с именем главной функции: все трансляторы настаивают на уникальности имени `main`.

Если для функций атрибут `extern` присваивается по умолчанию, то для переменных его надо прописывать явно. Рассмотрим простой пример:

```
  //  модуль m1.cpp
int x = 1;        //  определение переменной
void f(void);     //  объявление функции
int main()
```

```
{  f();    //  выводит 1
   x = 2;
   f();    //  выводит 2
   return 0;
}
  //  модуль m2.cpp
extern int x;    //  объявление переменной
void f(void)     //  определение функции
{ cout << x << endl; }
```

В модуле m1.cpp определена глобальная переменная `x`, которая используется функцией `f`, определенной в модуле m2.cpp. Во втором модуле переменная объявлена как внешняя (`extern`) — без этого возникают ошибки компоновки (повторное определение переменной). Причем, при объявлении переменной как `extern` ее нельзя инициализировать, т. к. в этом случае атрибут `extern` игнорируется. Как и для функций, атрибут `static` делает глобальную переменную локальной в модуле.

Пространства имен

Как мы уже упоминали в *гл. 2*, каждое имя в программе имеет некоторую область видимости (действия). Перечислим эти области видимости по возрастанию "объема".

- *Прототип.* Идентификатор, указанный в списке параметров прототипа функции, имеет областью действия только прототип функции.
- *Оператор.* Идентификатор, объявленный в условии оператора `if` или оператора цикла, действителен только до конца тела оператора.
- *Блок.* Имя, объявленное в блоке, является локальным в этом блоке и не видимо вне его.
- *Функция.* Только метка имеет такую область действия; в одной функции все метки должны быть различны.
- *Класс.* Элементы структур, объединений и классов видимы лишь в пределах класса.
- *Файл.* Ранее мы узнали, как имя может быть сделано видимыми в пределах файла.
- *Пространство имен.* C++ позволяет явно задать область видимости имен, присвоив области видимости некоторое имя.

Имена в одной области видимости не должны быть одинаковы, но в разных областях это вполне допустимо.

Именованные пространства имен

Использование атрибута `static` для того, чтобы сделать имя локальным в модуле — это устаревшая практика. В стандарте C++ определен другой механизм, который можно для этого применить — *пространство имен*. Пространства имен — относительно новый механизм C++ (по сравнению, например, с шаблонами и исключениями). Мы уже неоднократно пользовались стандартным пространством имен `std`. Однако язык предоставляет программисту возможность объявлять собственные пространства имен. Объявление пространства имен — это назначение имени для области видимости, в которую будут входить компоненты пространства имен. Общий синтаксис для объявления выглядит так:

```
namespace имя
{ //  объявления и определения }
```

Идентификатор `namespace` является зарезервированным словом. Объявления и определения могут включать описания переменных, функций, классов, типов, шаблонов и т. д. Как пишет Б. Страуструп [37, с. 211], пространства имен являются механизмом отражения логической структуры программы. Если некоторые объявления можно объединить по определенному критерию, то их следует поместить в одно пространство имен для отражения этого факта. Например, все имена, относящиеся к вводу/выводу, можно поместить в пространство имен `IO`.

Доступ к элементам пространства имен (в другой единице трансляции) выполняется при помощи операции разрешения контекста:

```
пространство_имен::компонент
```

Здесь идентификатор пространства имен служит квалификатором для имени компонента. Как мы уже упоминали, для стандартного пространства имен это выглядит, например, так:

```
std::cout
```

или так:

```
std::endl
```

Чтобы не писать имена с квалификаторами, можно использовать либо объявления со словом `using`:

```
using пространство_имен::компонент;
```

либо директиву со словом `using`:

```
using namespace пространство_имен;s
```

Слово `using` так же, как и `namespace`, является зарезервированным словом. Именованные пространства имен позволяют нам легко избавиться от воз-

можных конфликтов имен. Если два программиста работают над разными частями одной большой программы, то каждый из них может объявить собственное пространство имен (область видимости) и больше не беспокоиться о возможных повторных объявлениях. Главное, чтобы имена пространств имен не совпадали.

Однако короткие имена могут привести к конфликтам, а длинные программисты писать не любят. Б. Страуструп — такой же программист, как и мы с вами, поэтому его тоже заботила эта маленькая проблема. Как следствие, в стандарт C++ включили возможность объявлять *алиасы* (синонимы имен):

```
namespace DCS = Departament_of_Computer_Science;
```

Объявленное пространство имен разрешается разбивать на части. Если в разных единицах трансляции объявлено одно и то же пространство имен, то оно "склеивается" в единое пространство. Именно таким образом объявлено стандартное пространство имен `std`. Для реализации, по словам Б. Страуструпа [37, с. 228], мы должны "распределить" пространство имен по нескольким заголовочным файлам и файлам исходного кода:

```
  //  модуль m1.cpp
namespace A {
            void f1(void) {...};
            int x;
            }
  //  модуль m2.cpp
namespace A {
            void f2(void) {...};
            int y;
            class A{};
            }
```

Единственное ограничение — все имена в таком пространстве имен должны быть различны. Это мгновенно становится понятным, если мы напишем объединенное пространство имен:

```
namespace A {
            void f1(void) {...};
            int x;
            void f2(void) {...};
            int y;
            class A{};
            }
```

Такое ограничение естественно: т. к. имя пространства имен объявляет область видимости, это требование означает то, что уже нам известно — все имена в одной области видимости должны быть различны.

Мы прописали определения функций внутри пространства имен. Однако можно задавать реализацию и вне пространства имен, совершенно аналогично, как это делается для классов, например, так:

```
void A::f1(void) {...};
```

Пространства имен могут быть вложенными. В стандарте приведен такой пример:

```
namespace Outer {    //  внешнее пространство имен
int i;
namespace Inner {    //  вложенное пространство имен
                void f() { i++; }    //  работает Outer::i
                int i;
                void g() { i++; }    //  работает Inner::i;
                }
}
```

В справочной системе интегрированной среды Borland C++ Builder 6 показано, как задать алиас *внутреннего пространства имен*:

```
namespace BORLAND_SOFTWARE_CORPORATION {           //  внешнее
 /* тело namespace */
namespace NESTED_BORLAND_SOFTWARE_CORPORATION {    //  вложенное
 /* тело namespace */
                                                 }
                                              }
  //  алиасы namespace
namespace BI = BORLAND_SOFTWARE_CORPORATION;       //  внешнее
  //  внутреннее, задается с квалификатором
namespace NBI =
       BORLAND_SOFTWARE_CORPORATION::NESTED_BORLAND_SOFTWARE_CORPORATION;
```

Как видим, для задания алиаса внутреннего пространства имен указывается квалификатор — имя внешнего алиаса.

Неименованные пространства имен

Если имена определены вне всех пространств имен, на самом верхнем уровне, то они входят в *глобальное пространство имен*. Чтобы обратиться к компоненту глобального пространства имен, можно воспользоваться операцией разрешения контекста `::`. Это необходимо в тех случаях, когда существуют совпадающие идентификаторы в локальном и глобальном пространстве имен, т. к. по умолчанию всегда выбирается переменная с наименьшей областью видимости. Например, таким способом можно обратиться к именам, определенным в заголовочных файлах Windows (например, в windef.h):

```
::max(5, 6)
```

Для локализации имени в файле вместо атрибута `static` в C++ разрешается задавать *анонимные* (неименованные) пространства имен. Анонимные пространства имен удобно называть *локальными пространствами имен*, т. к. они по сути своей таковыми и являются. В справочной системе интегрированной среды Borland C++ Borland 6 приводится пример, который был немного модифицирован и выполнен в интегрированной среде Visual C++ 6. Текст примера приведен в листинге 11.9.

Листинг 11.9. Локальные пространства имен

```
  //  модуль m1.cpp
#include <iostream>
extern void func(void);     //  функция в этом модуле не определена
namespace {                 //  анонимное пространство имен
          float pi = 3.1415926F;    //  pi доступно только в этом файле
          }
int main() {
  //       float pi = 0.1;      //  1:локальное определение того же имени
           std::cout << "pi = " << pi << std::endl;
           func();
           return 0;
           }
  //  модуль m1.cpp
#include <iostream>
namespace {                     //  анонимное namespace
          float pi = 10.0001F;  //  pi доступно только в этом файле
          void f(void) {        //  определение локальной функции
          std::cout << "First func() called; pi = " << pi<<std::endl;
                    }
          }
void func(void) {               //  определение глобальной функции
                f();
                std::cout << "Second func() called; pi = " << pi;
                }
```

Эта программа выведет на экран:

```
pi = 3.14159
First func() called; pi = 10.0001
Second func() called; pi = 10.0001
```

В модуле m1.cpp определено локальное пространство имен, в котором задана константа `pi`. Если в функции `main` строка `1` закомментирована, то выводится `pi` из локального пространства имен. Если же комментарий удалить,

то в операторе вывода будет использоваться имя с более ограниченной областью видимости (имя `pi` в блоке — теле главной функции). В этом случае мы никаким способом не сможем добраться до имени, определенном в локальном пространстве имен.

В файле m2.cpp определено свое локальное пространство имен, в котором прописано то же имя `pi`, но задана другая константа. В том же локальном пространстве имен определена локальная в m2.cpp функция `f()`. Вне локального пространства определена глобальная функция `func()`, которая вызывает локальную функцию `f()`. В модуле m1.cpp прописан прототип глобальной функции, чтобы ее можно было вызвать в этом модуле. В файле m1.cpp невозможно использовать ни функцию `f()`, ни имя `pi` из файла m2.cpp.

Использование препроцессора

При раздельной трансляции и последующей компоновке требуется, чтобы все объявления были согласованы. Простым, но ставшим стандартом "де-факто", методом достижения согласованности объявлений в различных единицах трансляции является включение заголовочных файлов в исходные файлы. Заголовочные файлы содержат информацию об интерфейсах (например, прототипы функций), а исходные файлы — исполняемый код и/или определения данных. Включение выполняется препроцессором по директиве `#include`. Препроцессор, обрабатывая директиву

```
#include "включаемый_файл"
```

заменяет эту строку на содержимое указанного файла. Этот файл должен содержать правильный текст на C++, т. к. его будет обрабатывать компилятор. Указанный файл должен находиться в текущем каталоге. Для включения стандартных библиотечных файлов используется уже известная нам форма:

```
#include <включаемый_файл>
```

Указанный заголовочный файл должен находиться в стандартном каталоге интегрированной среды include. Пробелы внутри угловых скобок или внутри кавычек — значимые, поэтому без необходимости дополнительные пробелы писать нельзя.

Так как при трансляции разных частей программы часто необходимы одни и те же включаемые файлы, то при объединении частей в полную программу возникает избыточность объявлений, что может привести к ошибкам трансляции. Для решения этой проблемы опять используется препроцессор: мы должны прописать во включаемом файле "стража" включения. Для этого используется директива `#define` и директивы условной трансляции. Обычно это делается так:

```
  //  myfile.h
#ifndef MYFILE    //  "страж" определен?
#define MYFILE    //  нет — определяем
  //  содержимое файла
#endif
```

Когда препроцессор обрабатывает первую директиву,

```
#include "myfile.h"
```

имя `MYFILE` ("страж") не определено. Поэтому содержимое файла включается в обработку. Если эта директива встречается еще раз, то "страж" уже определен и содержимое файла пропускается. Именно так организованы стандартные заголовочные файлы. Например, начало стандартного включаемого файла math.h в интегрированной среде Borland C++ 5 выглядит так:

```
/* math.h
  Definitions for the math floating point package.
*/
/*
*  C/C++ Run Time Library — Version 8.0
*
*  Copyright (c) 1987, 1997 by Borland International
*  All Rights Reserved.
*
*/
/* $Revision: 8.10 $ */
#ifndef __MATH_H
#define __MATH_H
```

Как видите, определен "страж" с именем `__MATH_H`. Соответствующий файл системы Visual C++ 6 практически не отличается:

```
/***
*math.h — definitions and declarations for math library
*
* Copyright (c) 1985-1997, Microsoft Corporation. All rights reserved.
*
*Purpose:
*    This file contains constant definitions and external subroutine
*    declarations for the math subroutine library.
*    [ANSI/System V]
*
*    [Public]
*
****/
```

```
#if _MSC_VER > 1000
#pragma once
#endif

#ifndef _INC_MATH
#define _INC_MATH
```

В этой системе определен "страж" с именем `_INC_MATH`.

По негласному практическому правилу включаемые файлы обычно содержат определения и/или объявления констант, типов, шаблонов, функций, перечислений, поименованных пространств имен, макросов, директив условной компиляции, комментариев. Изучение стандартных заголовочных файлов может дать много информации об этом.

ГЛАВА 12

Обобщенное программирование

По стандарту шаблон определяет семейство классов или функций. Мы уже неоднократно использовали шаблоны именно в этом качестве — как параметризованные классы и параметризованные функции. Однако концепция шаблонов в C++ гораздо глубже, чем простая параметризация типа. Например, шаблоны позволяют перевести некоторые вычисления на этап компиляции программы. Этот прием получил название *метавычисления*. А перегрузка операции () в сочетании с шаблонами позволяет максимально обобщить программирование традиционных алгоритмов вроде сортировки и поиска. Блестящим подтверждением этого является стандартная библиотека шаблонов STL.

Шаблоны — это механизм, который был разработан и включен в C++ одним из последних перед утверждением стандарта [38]. Первоначально программистское сообщество фактически не понимало их действительную роль в развитии C++ и всего программирования в целом. Однако сейчас, по прошествии 10 лет, совершенно ясно, что никакие другие свойства языка не оказали на программирование такого влияния, как шаблоны. В [37] Б. Страуструп описывает различные *парадигмы программирования*, которые поддерживает C++: *процедурное программирование*, *модульное программирование* и *объектно-ориентированное программирование*. С появлением шаблонов программирование обрело еще один облик — *обобщенное программирование*.

Собственно, программисты всегда стремились писать программы как можно более универсальным способом — библиотеки стандартных математических подпрограмм появились "во второй день творения", а языки программирования — на "третий день". Поэтому обобщенное программирование имеет свою историю. Мне представляется, что обобщенное программирование тоже можно классифицировать аналогично парадигмам программирования. Процедурным обобщенным программированием естественно назвать процедурное программирование, при котором в качестве параметров функций

применяются указатели на функции. Совершенно очевидно, что объектно-ориентированное обобщенное программирование — это программирование паттернов [10] и реализация обобщенных структур данных — контейнеров. Программирование с использованием *функторов* и шаблонов можно назвать функционально-объектным программированием. А перенос вычислений на этап компиляции называется *метапрограммированием*.

Процедурное обобщенное программирование

В *гл. 4* мы выяснили, что функцию можно передавать в качестве параметра (см. листинг 4.18), и неоднократно пользовались этим при обработке динамических структур данных *(см. гл. 7)*. Имя функции, так же как и имя массива, является константой-указателем. Поэтому фактически мы передаем параметр-указатель в функцию, а не саму функцию. И совершенно аналогично типизированным указателям в C++ имеется возможность объявлять переменные-указатели на функцию. Определение переменной-указателя на функцию похоже на объявление прототипа, но отличается наличием звездочки:

```
тип (*имя указателя)(спецификация параметров);
```

Простой пример

```
int (*fp)(int);
```

определяет указатель `fp` на функцию, с параметром типа `int` и возвращающую результат типа `int`. Скобки `()` указывать обязательно, поскольку без них

```
int *fp(int);
```

получается совершенно другая по смыслу конструкция — функция с параметром типа `int`, возвращающая указатель на `int`.

Указателю на функцию можно присваивать значение — имя конкретной функции; при этом прототипы должны полностью совпадать. Отличаются лишь имена. Пример:

```
void f1(void){ };     //  одна функция
void f2(void){ };     //  другая функция
void (*pf)(void);     //  указатель на функцию
pf = f1;              //  присвоение имени функции f1
pf = f2;              //  присвоение имени функции f2
```

Как всякую переменную, указатель на функцию можно инициализировать. Это делается так:

```
void f1(void){ };          //  одна функция
void (*pf)(void) = f1;     //  инициализация
```

Вызов функции по указателю имеет следующий вид:

```
(*имя указателя)(список аргументов);
```

Разрешается и более простой вызов:

```
имя указателя(список аргументов);
```

Например, вызов указанных выше функций по указателю `pf` выглядит так:

```
(*pf)();
```

или так:

```
pf();
```

Указатели на функцию, как и всякие другие переменные, могут быть объявлены статическими. Через статический указатель можно выполнять рекурсивный вызов, как приведено в листинге 12.1 (Visual C++ 6).

Листинг 12.1. Рекурсивный вызов функции через указатель

```
void f(int k)
{  static void (*g)(int k) = f;    //  объявление и инициализация
   if (k > 0)    //  условие продолжения
   {  cout << "call " << k << endl;
      g(k-1);    //  рекурсивный вызов
      cout << "return" << k << endl;
   }
}
```

Функция написана в третьей рекурсивной форме, выполняет действия на рекурсивном спуске и на рекурсивном возврате. При вызове `f(3)` на экран выдается:

```
call 3
call 2
call 1
return 1
return 2
return 3
```

При объявлении указателя как локальной (не статической) переменной наблюдается та же картина.

Поскольку указатели на функции — это полноценные переменные, язык C++ разрешает объявлять массивы указателей на функции. Однако сделать

это, не применяя оператор typedef, достаточно сложно. Во всяком случае не сразу удастся. Правильное объявление выглядит так:

```
float (*p[5])(float);
```

Вызов функции посредством указания элемента массива демонстрирует программа (Visual C++ 6), текст которой приведен в листинге 12.2.

Листинг 12.2. Массив указателей на функции

```
double f(double)                // определение "нашей" функции
{ cout << 'f' << endl; return 0; }
int main(void)
{  double (*p[3])(double);      // массив указателей на функцию
   p[0] = sin;
   p[1] = cos;                  // стандартные функции
   p[2] = f;                    // определенная нами функция
   cout << p[0](1) << endl;     // вызывает sin(1.0)
   cout << p[1](1) << endl;     // вызывает cos(1.0)
   cout << p[2](1) << endl;     // вызывает f(1.0)
   return 0;
}
```

На экран программа выводит результаты, не требующие комментариев:

```
0.841471
0.540302
f
```

Поскольку явное объявление указателя на функцию обычно бывает достаточно длинно, для сокращения записи используют оператор typedef. Синтаксис применения typedef при определении синонима для указателя на функцию, так же как и для массива, не похож на стандартный:

```
typedef <возвращаемый функцией тип>(*<новый_тип>)(<список параметров>);
```

Таким образом, определение

```
typedef float (*PF)(float);
```

вводит новое имя типа PF, использовать которое можно так:

```
PF p1;       //  один указатель
PF p2[5];    //  массив указателей на функцию float имя(float)
```

А теперь — упражнение для любителей головоломок. Попробуйте без оператора typedef написать прототип функции, получающей параметр типа int, возвращающей указатель на функцию следующего вида:

```
int(*function)(int *)
```

Я уверен, что это у вас не получится. При использовании `typedef` это довольно просто:

```
typedef int (*function)(int *);
function t(int);
```

Без применения оператора `typedef` объявление выглядит почти "китайской грамотой":

```
int (*t(int))(int*);
```

Принцип такой (от обратного) — на месте имени `function` в операторе `typedef` нужно поставить имя и список параметров функции, возвращающей указатель на функцию. Напишем прототип функции `tt`, возвращающей указатель на функцию типа `int(*f)(int, int*)` и имеющей список параметров `(int*, int)`. Судя по предыдущему примеру, это должно выглядеть так:

```
int (*tt(int*, int))(int, int*)
```

Visual C++ 6 не имеет ничего против.

Имена, объявленные в операторе `typedef`, подчиняются обычным правилам блочной области видимости. Это значит, что имя, прописанное внутри функции `typedef`, "не работает" снаружи — при попытке использовать объявленное имя вне тела функции транслятор выдает обычные сообщения о необъявленном идентификаторе. Как и при объявлении переменных, внутренний оператор `typedef` скрывает внешний. Небольшой пример иллюстрирует эту ситуацию:

```
void f(void)
{ typedef int (*t)(int); }     //  виден только внутри
int main(void)
{ t a; }                       //  ошибка — имя t здесь не определено
```

Полиморфные функции

Указатели на функции используются для программирования *полиморфных подпрограмм*, т. е. подпрограмм, которые могут обрабатывать данные различных типов. В стандартную библиотеку stdlib.h входит несколько функций, принимающих в качестве параметра указатель на функцию сравнения: линейный поиск в массиве, двоичный поиск, сортировка массива. Сортировка имеет следующий прототип:

```
void qsort(void *base,      //  адрес массива
           size_t num,      //  количество элементов
           size_t width,    //  размер элемента
           int (*compare)(const void *elem1, const void *elem2));
```

Естественно, функция реализована как универсальная, способная работать с массивами любых типов. Поэтому массив передается в функцию как указатель `void *`. Так как данных типа `void` в C++ нет, то функция сортировки сама не может сравнивать элементы массива. Поэтому необходима функция сравнения (элементов массива), которую функция сортировки получает в качестве параметра и вызывает по указателю. В соответствии с принятыми соглашениями такая функция должна сравнивать два элемента и возвращать `1`, `0` или `-1` в зависимости от результата сравнения. Б. Страуструп в [37] приводит пример определения таких функций, рассматривая применение универсальной функции сортировки. Пусть у нас есть массив структур:

```
struct User
{  char *name;     //  название отдела
   int dept;       //  номер отдела
}
User V[100];
```

Определение функции сравнения для сортировки этого массива по названиям будет таким:

```
int cmpS(const void* m, const void* n)
{ return strcmp(static_cast<const User*>(m)->name,
         static_cast<const User*>(n)->name);
}
```

Определение функции для сравнения номеров отделов выглядит так:

```
 int cmpD(const void* m, const void* n)
{ return (static_cast<const User*>(m)->dept    //  вычитаем
          static_cast<const User*>(n)->dept);
}
```

Эта функция возвращает не "канонические" `-1`, `0`, `1`, но сортировка, естественно, работает. Преобразование типов, как мы знаем, делать обязательно, но можно сделать это проще — в стиле C: `((User *)m)`.

Сортировка по названиям выполняется так:

```
qsort((void *)V, 100, sizeof(User), cmpS);
```

А для сортировки по номерам вызов должен быть таким:

```
qsort((void *)V, 100, sizeof(User), cmpD);
```

Обратите внимание на то, что во всех приведенных примерах указатель на функцию передается по значению. Это обычная практика. Более того, алгоритмы стандартной библиотеки принимают параметры-указатели на функции именно по значению, хотя по правилам языка C++ разрешается передавать указатель на функцию и по ссылке, и по указателю.

Способы передачи параметров-указателей на функции

С использованием определенного имени `PF` прототип функции, вычисляющей корень уравнения `f(x) = 0` на отрезке `[a, b]`, выглядит так:

```
double R(PF f, double a, double b, double epsilon)
```

Здесь `a` и `b` — отрезок локализации корня, `epsilon` — точность вычисления. Для вычисления корня функции $y = x3 - 1.5x + 1.2$ на интервале $[0, 2]$ с точностью 10^{-5} необходимо лишь написать следующую программу:

```
float f(float x)    //  определение функции
{ return (x*x*x - 1.5*x + 1.2); }
int main(void)
{ double result = R(f, 0.0, 2.0, 0.000001); }
```

Параметрам-указателям на функции можно присваивать значения по умолчанию. Для функции, вычисляющей корень, это может выглядеть так:

```
double R(PF f = cos, double a = 0.0, double b = 2.0,
                double epsilon = 10e-5)
```

Тогда вызов сильно сократится:

```
double y = R();
```

Передача указателя на функцию по ссылке или по указателю выглядит традиционно:

```
double R(PF &f, double a, double b, double epsilon);    //  по ссылке
double R(PF *f, double a, double b, double epsilon);    //  по указателю
```

Однако если при передаче по значению и по ссылке вызов передаваемой функции может быть указан как `f(параметры)`, то при вызове по указателю прописывать звездочку обязательно:

```
(*f)(параметры);
```

Как и другие типы параметров, указатель на функцию может быть константным параметром, например:

```
double R(const PF &f, double a, double b, double epsilon);
```

Указатели на функцию в списках переменной длины

Все описанное ранее — довольно простые вещи, которые при некоторых усилиях можно найти в различных книжках по C++. Однако ни в одной книге не упоминаются более интересные вопросы.

1. Можно ли задать указатели на функцию в переменном списке параметров?

2. Как выполнять преобразование типа с указателями на функции?
3. Может ли функция возвратить указатель "на себя", и где это может пригодиться?

Ответ на первый вопрос очевиден: список параметров может состоять из указателей на функции. Осталось научиться это делать. Давайте рассмотрим простой пример обработки массива несколькими функциями. Сначала надо решить, какой прототип будет у функций обработки. Прототипы всех функций должны быть одинаковы. Пусть функция получает массив и размер как параметры, например:

```
void f(double m[], int n)
```

В качестве последнего обязательного параметра в списке естественно указать первую обрабатывающую функцию. Организация вызовов функций, очевидно, сложностей не представляет: в цикле до `NULL` выполняется вызов по указателю на функцию. Однако перебор указателей в списке — это проблема. Дело в том, что с указателями на функции нельзя выполнять никаких арифметических действий. Кроме того, как и в случае передачи "простых" типизированных указателей, здесь требуется "двойной" указатель на указатель, чтобы перемещение по списку было правильным. Так, без "обмана" компилятора и преобразований типа не обойтись. Пользуясь тем, что для микропроцессора Intel размер любого указателя равен 4 байта, это можно сделать. Конечно, при переходе на другую платформу могут возникнуть проблемы, однако на платформе Intel все работает правильно. Необходимо только следить за наличием и отсутствием звездочек в соответствующих местах. Программа, текст которой приведен в листинге 12.3, корректно отрабатывает и в Visual C++ 6, и в Borland C++ 5.

Листинг 12.3. Полиморфная функция с переменным списком параметров

```
void f1(double b[], int k)          //  функция обработки
{ for (int i = 0;i < k; i++)  b[i] += 1; }
void f2(double b[], int k)          //  функция обработки
{ for (int i = 0;i < k; i++) b[i] *= 2; }
void F(double a[], int n, void (*f)(double m[], int n)...)
{  typedef void (*pf)(double m[], int n);
   double **pp = (double **)&f;     //  первый "обман"
   while (*pp != NULL)              //  значение NULL
   {  pf *p = (pf*)pp;              //  второй "обман"
      (*p)(a, n);                   //  звездочка обязательна
      pp++;                         //  перемещение по списку
   }
}
int main(void)
```

```
{  double V[10];
   for (int i = 0; i < 10; ++i) V[i] = i;
   for (i = 0; i < 10; ++i) cout << V[i] << ' ';
   cout << endl;
   F(V, 10, f1, f2, 0);              //  вызов "главной" функции
   for (i = 0; i < 10; ++i) cout << V[i] << ' ';
   return 0;
}
```

Функции обработки просты: первая просто увеличивает значения элементов массива на 1, а вторая удваивает их. Программа выведет на экран две строки:

```
0 1 2 3 4 5 6 7 8 9
2 4 6 8 10 12 14 16 18 20
```

При исследовании функции `F` необходимо обратить внимание на несколько моментов:

- в операторе `double **pp = (double **)&f` тип указателя на указатель нам не важен, поскольку любой указатель имеет размер 4 байта. Кроме того, операцию получения адреса `&` указывать обязательно, т. к. указатель на функцию — однократный, а изменяем мы двойной указатель;
- `NULL` является значением, а не адресом, поэтому в цикле пишется `*p` с одной звездочкой;
- мы вынуждены второй раз "обманывать" транслятор, поскольку `pp` не является указателем на функцию. Оператор `pf *p = (pf *)pp` совершенно необходим — он преобразует "двойной" типизированный указатель в "двойной" указатель на функцию;
- по причине "двойственности" указателя `p` необходимо писать "косвенный" вызов `(*p)(a, n)`. Звездочка обязательна, т. к. "простой" указатель `p` не является указателем на функцию — без нее программа не транслируется;
- мы "бежим" по списку указателей, наращивая "двойной" типизированный указатель.

Разрабатывая данную программу, мы ответили и на второй вопрос утвердительно. Можно написать преобразования указателей в стиле последнего стандарта C++:

```
double **pp = reinterpret_cast<double **>(&f);
pf *p = reinterpret_cast<pf *>(pp);
```

Используя явные приведения типов, мы можем написать функцию с переменным числом параметров, элементами списка параметров которой будут

указатели на функции разного типа. Однако, как вы понимаете, чудес все-таки не бывает: функция обработки не может самостоятельно определить типы передаваемых указателей во время выполнения. Следовательно, она должна получать в качестве параметра некоторый массив, в котором прописана последовательность типов указателей на функции. Этот параметр как раз и может быть последним обязательным параметром в списке.

При разработке функции с переменным числом параметров разного типа первое, что должен сделать программист — определиться с типами параметров-функций. Покажем реализацию данной полиморфной функции с помощью стандартных макросов `va_...` Определим типы указателей на функции:

```
enum typeFunc { type1 = 1, type2 = 2, end = 0 };
```

Сами функции теперь будут, естественно, разного типа:

```
void f1(double b[], int k, int n)   //  увеличивает элементы массива на n
{ for (int i = 0; i < k; i++) b[i] += n; }
void f2(double b[], int k)           //  удваивает элементы массива
{ for (int i = 0; i < k; i++) b[i] *= 2; }
```

В первой функции мы добавили параметр — целое значение, на которое будут увеличиваться элементы массива. Как было решено, после обрабатываемого массива в функцию передается массив типов функций, в конце которого стоит ноль. Массив типов используется оператором-переключателем, а последний ноль служит признаком для окончания цикла. Текст полиморфной функции с переменным списком параметров приведен в листинге 12.4.

Листинг 12.4. Полиморфная функция со списком параметров разных типов

```
void ff(double a[], int n, typeFunc tp[],...)
{  //  определение типов указателей
   typedef void (*pf1)(double m[], int n, int k);
   typedef void (*pf2)(double m[], int n);
   va_list p;                    //  определение указателя на список
   va_start(p, tp);              //  установка указателя
   for (int i = 0; tp[i]; i++)   //  пока тип != нулю
   {  switch(tp[i])
      {  case type1:                    //  вызов функции типа 1
         {  pf1 pp = va_arg(p, pf1);    //  получили указатель
            pp(a, n, 3);                //  вызов функции
            break;
         }
         case type2:                    //  вызов функции типа 2
```

```
         {  pf2 pp = va_arg(p, pf2);    //  получили указатель
            pp(a, n);                   //  вызов функции
            break;
         }
      }
   }
   va_end(p);    //  "закрыли" указатель
}
```

Как видно, использование макросов упрощает работу с указателями — нет нужды "обманывать" компилятор. Интересно, что объявлять указатель на функцию нет необходимости — совершенно правильно работает и такой вызов:

```
(pf1(va_arg(p, pf1)))(a, n, 3);
```

Вызов полиморфной функции требует сначала заполнения массива типов:

```
typeFunc tp[3] = { type1, type2, end };    //  массив типов функций
```

Тогда вызов выглядит так:

```
ff(V, 10, tp, f1, f2);    //  вызов
```

`NULL` в конце ставить нет необходимости, т. к. в массиве типов стоит ноль-индикатор. Следующая программа

```
int main(void)
{  double V[10];              //  масиив
   for (int i = 0; i < 10; ++i)
   V[i] = i;                  //  заполнение
   for (i = 0; i < 10; ++i)
   cout << V[i] << ' ';       //  вывод исходного массива
   cout << endl;
   typeFunc tp[3] = { type1, type2, end };    //  массив типов функций
   ff(V, 10, tp, f1, f2);     //  вызов
   for (i = 0; i < 10; ++i)
   cout << V[i] << ' ';       //  вывод массива-резульатата
   cout << endl;
   return 0;
}
```

выводит на экран две строки:

```
0 1 2 3 4 5 6 7 8 9
6 8 10 12 14 16 18 20 22 24
```

Вообще-то именно для случаев такого полиморфизма и были придуманы перегрузка функций и шаблоны, которые переводят подобные проблемы с этапа выполнения на этап компиляции.

Возврат указателя на функцию

Ну а теперь нам осталось рассмотреть возврат указателя на функцию. В общем-то, единственной проблемой в этом случае является синтаксис — без использования оператора `typedef` написать прототип функции, возвращающей указатель на функцию, достаточно сложно. Но мы не будем заниматься синтаксическими изысками, а рассмотрим "предельный вариант": возврат функцией указателя "на себя". В [35] приведена идея использования данной возможности: Герб Саттер пишет, что можно применять функции, возвращающие указатель "на себя" для реализации конечных автоматов.

С помощью конечных автоматов можно решать огромное количество разнообразнейших задач. Даже классический рекурсивный алгоритм "Ханойские башни" можно реализовать в виде некоторого автомата [47]. Конечные автоматы широко применяются в компиляторах для распознавания простых конструкций языка: идентификаторов, зарезервированных слов, чисел различного формата. Поэтому знать основы реализации конечных автоматов очень полезно.

С теоретической точки зрения распознающий конечный автомат представляет собой устройство, на вход которой поступает строка символов. Устройство, прочитав очередной символ, переключается в некоторое состояние. Какое — зависит от символа. Одно из состояний является завершающим. Если автомат перешел в это состояние, то его работа прекращается, причем считается, что входная строка является правильной. Одно из состояний является ошибочным: переход автомата в это состояние означает наличие "неправильного" символа во входной строке. На бумаге конечные автоматы обычно изображаются в виде графа, вершинами которого являются состояния, а дуги обозначены соответствующими символами.

Для демонстрации реализации конечного автомата функциями, возвращающими указатель "на себя", мы рассмотрим простейший автомат, распознающий идентификатор *(см. гл. 1)*. Автомат представлен на рис. 12.1. У автомата 4 состояния: "Старт", "Следующая буква", "Ошибка" и "Финиш". Программа начинает работу в состоянии "Старт". Если первый символ — буква, то происходит переход в состояние "Следующая буква". В этом состоянии программа пребывает до тех пор, пока в строке не встретится символ, недопустимый для идентификатора — тогда происходит переход в состояние "Финиш". Переход в состояние "Ошибка" выполняется только в том случае, если первый символ — не буква. Автомат намеренно упрощен (в частности, не отслеживается допустимая длина идентификатора) с целью сосредоточения именно на существенных вещах: переключениях между состояниями.

Сначала реализуем автомат традиционным способом — с использованием оператора-переключателя. Текст программы приведен в листинге 12.5 (не указаны операторы `#include` и `namespace`). Программа не требует особых по-

яснений, поскольку практически "в лоб" реализует описанный конечный автомат.

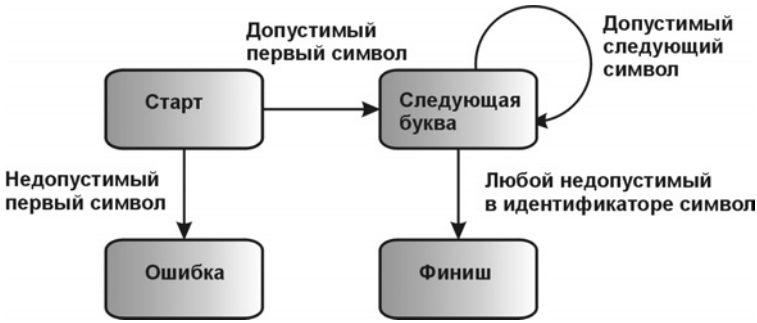

Рис. 12.1. Конечный автомат, распознающий идентификатор

Листинг 12.5. Конечный автомат — распознаватель идентификатора

```
int main(void)
{  string s = "i123456789";     //  проверяемая строка
   enum States                  //  состояния
   {  Start,
      NextLetter,               //  следующая буква
      Error,                    //  ошибка
      Finish
   };
   States x = Start;            //  начальное состояние
   int i = 0;
   while((x != Finish) && (x != Error)) //  выход по ошибке или нормально
   {  switch(x)                         //  переключатель состояний
      {  case Start:
         if (isLetter(s[i])){ i++; x = NextLetter; }
         else x = Error;
         break;
         case NextLetter:
         if (isNextLetter(s[i])) { i++; x = NextLetter; }
         else x = Finish;
         break;
         case Error:
         cout << s << "Неправильный начальный символ!" << endl;
         break;
      }
   }
```

```
   cout << s << endl;
   return 0;
}
```

Реализация функций `isLetter` и `isNextLetter` тривиальна, поэтому оставляем это в качестве небольшого упражнения. Цикл работает либо до ошибки (первый символ — не буква или подчеркивание), либо до первого недопустимого в идентификаторе символа — например, пробела.

Теперь нам необходимо преобразовать программу в другую — с функциями, возвращающими указатель "на себя", — но реализующую те же действия. Во-первых, сначала нужно написать правильный `typedef`, что само по себе — не простая проблема: ведь определение типа "указатель на себя" рекурсивно — слева от определяемого типа указателя должен стоять тот же тип:

```
typedef function (*function)(...);
```

В C++ такие определения недопустимы — придется "обманывать" компилятор:

```
typedef void (*fp)(const char ch);
typedef fp (*States)(const char ch);
```

Это еще не все. Как мы видим по второму оператору `typedef`, состояния в нашей новой программе будут представлены функциями. Параметром у функций-состояний является символ строки-идентификатора. Тогда переход в новое состояние — это оператор `return` с именем функции-состояния в качестве выражения. Текст, преобразованной программы приведен в листинге 12.6.

Листинг 12.6. Конечный автомат — набор функций

```
fp Finish(const char ch)
{ return (fp)NULL; }     //  для окончания цикла
fp Error(const char ch)
{  cout << ch << "Неправильный начальный символ!" << endl;
   return (fp)NULL;      //  для окончания цикла
}
fp NextLetter(const char ch)
{  if (isNextLetter(ch)) return (fp)NextLetter;
   else return (fp)Finish;
}
fp Start(const char ch)
{  if (isLetter(ch)) return (fp)NextLetter;
   else return (fp)Error;
}
```

```
int main(void)
{  string s ="i12345.";
   States x = Start;              //  стартовое состояние
   int i = 0;
   while(x != NULL)
   { x = (States)x(s[i++]); }     //  переход в новое состояние
   cout << s << endl;
   return 0;
}
```

В этой программе нужно обратить внимание на следующие особенности:

- в операторе `return` все-таки приходится еще раз "обманывать" компилятор, применяя явное преобразование типов — без этого невозможно реализовать возврат "на себя";
- аналогичный обратный "обман" делается в операторе `x = (States)x(s[i++])`, который является вызовом функции посредством указателя;
- пользуясь тем, что указатель может быть нулевым, мы возвращаем `NULL` из функций `Finish` и `Error`. В данном случае функция `Finish` ничего не делает, просто возвращая условие завершения цикла. Очевидно, что на эту функцию можно "навесить" (как, впрочем, и на все остальные функции) любое необходимое действие, например, запись идентификатора в таблицу имен.

В программе мы неоднократно пользуемся непереносимыми преобразованиями указателей. Однако, на мой взгляд, это совершенно ничтожная плата за возможность кардинального "вырождения" тела цикла. Не секрет, что тело оператора `switch` при реализации сложных (в частности, вложенных) автоматов бывает очень большим. При любом изменении конечного автомата (добавление и удаление состояний) придется модифицировать большой оператор-переключатель, что чревато ошибками.

Вторая программа (см. листинг 12.6) — совсем другое дело. Аналогичные изменения в ней производятся удалением или добавлением функций. Возможность ошибки существенно снижается. Инкапсуляция, следовательно, корректность и надежность программы значительно возрастают. Таким образом, возврат функцией указателя "на себя" существенно упрощает жизнь программисту.

Функционально-объектное обобщенное программирование

Как мы убедились, указатели на функции — это достаточно мощное средство C++. Все стандартные алгоритмы реализованы так, что способны принимать указатель на функцию в качестве параметра. Однако, как мы уже

упоминали в *гл. 5*, в стандартной библиотеке реализованы стандартные объекты-функции, которые также называют функциональными объектами. Мы будем пользоваться и тем и другим термином.

Объект-функция — это экземпляр класса, в котором перегружена операция вызова функции `()`. Класс, в котором перегружена операция вызова функции, будем называть *функциональным классом*. Шаблон объекта-функции (функционального класса) часто называют *функтором*.

Перегрузка операции вызова позволяет использовать объект-функцию так же, как обычную функцию. С другой стороны — это полноценный объект, который можно передавать как параметр (любым способом) и получать в качестве результата. Объекты-функции обычно применяются при обращении к стандартным алгоритмам. Вспомним пример поиска в списке (см. листинг 5.16) и преобразуем функцию-предикат в функцию-объект (текст функции-объекта приведен в листинге 12.7).

Листинг 12.7. Использование функции-объекта

```
struct isPrime {
bool operator() (int number) const
{  number = abs(number);
   if (number == 0 || number == 1) return false;
   int divisor;
   for (divisor = number/2; number%divisor != 0; --divisor);
   return divisor == 1;
}
};
```

Функцию-объект можно использовать так:

```
list<int> L;
list<int>::iterator pos;                              //  итератор
  //  заполняем список целыми числами от 20 до 40
for (int i = 20; i <= 40; ++i) L.push_back(i);
pos = find_if (L.begin(), L.end(), isPrime());     //  ищем простое число
if (pos != L.end()) cout << *pos << " — первое простое число" << endl;
else cout << " — нет простого числа" << endl;
  //  считаем простые числа в списке
cout << count_if(L.begin(), L.end(), isPrime()) << endl;
```

Пока это мало чем отличается от использования простой функции-предиката. Однако объекты-функции обладают всеми возможностями указателей на функции и имеют еще массу других достоинств. Так как операция вызова функции может возвращать значение любого типа и получать произвольное количество параметров, перегрузка этой операции позволяет реали-

зовать любую необходимую функциональность. Имя функционального класса можно использовать в качестве параметра шаблона другого класса, что позволяет избавиться от накладных расходов вызова функции при выполнении программы. Функциональные классы могут участвовать в иерархии наследования, поэтому операцию вызова функции можно сделать виртуальной. Это позволяет нам избавиться от многих операторов-переключателей вроде тех, что пришлось писать в примере обработки массива функциями разных типов (см. листинг 11.5). Виртуальность позволяет нам реализовать и *мультиметоды* [1, 34, 49]. В функциональном классе можно объявить поля, что позволит нам иметь функциональный объект с состояниями. Также в функциональном классе может быть объявлен конструктор инициализации, который будет инициализировать объект, считывая информацию из файла на диске. Открывающиеся возможности столь велики, что про функциональные классы, объекты и шаблоны можно написать отдельную очень большую книгу "Функционально-объектное программирование".

Но вернемся "с небес на землю" и рассмотрим более простые возможности, открывающиеся при использовании функциональных объектов. Предположим, мы пишем программу, предназначенную для отдела кадров. Отделу кадров регулярно требуется формировать разнообразные списки сотрудников (например, всех сотрудников одного отдела или всех сотрудников, у которых три и более детей). Сотрудник предприятия может быть представлен структурой или классом `Person` *(см. разд. "Закрытое наследование" гл. 11).*

```
class Person {
public:...
private:
string name;         //  фамилия
Date date;           //  дата поступления на работу
Address address;     //  адрес
int dep;             //  номер отдела
int children;        //  количество детей
bool sex;            //  пол
};
```

Множество сотрудников предприятия представлено вектором объектов:

```
vector<Person> V(100);
```

Нам требуется написать функцию-фильтр, формирующую новый вектор из исходного, выбирая элементы, удовлетворяющие условию отбора. Очевидно, нам потребуется три функциональных объекта-предиката: меньше, равно, больше. Однако аргументом этих предикатов может быть почти любое поле класса `Person`, поэтому нам нужны шаблоны. Реализация шаблонов-предикатов выглядит совершенно одинаково, поэтому приведем текст только одного из них — `lessthan` (листинг 12.8).

Листинг 12.8. Реализация предиката `lessthan` сравнения даты

```
template <class type_elem>
class lessthan {
public:
less(type_elem v) : v_(v) {}    //  конструктор
bool operator() ( type_elem v)
{ return v < v_; }
private:
type_elem v_;
};
```

Другие предикаты отличаются только названиями (`equalsto` — равно, `greaterthen` — больше) и операцией сравнения в теле операции вызова функции.

Примечание

Следите за тем, чтобы имена ваших предикатов не совпадали со стандартными, которые реализованы в библиотеке STL.

Операцию сравнения должен обеспечивать класс, элементы которого сравниваются.

Тогда прототип шаблона функции-фильтра будет такой:

```
template <class T>
vector <Person> filter(vector <Person> &v, lessthan<T>(x));
```

Реализацию оставляем в качестве упражнения.

Так как другие предикаты — это объекты другого типа, нам требуется перегрузка. Но мы не будем этого делать, а поступим по-другому — еще более обобщим функтор, передавая ему унарный объект-предикат в качестве второго параметра шаблона. Более того, мы можем присвоить этому параметру значение по умолчанию: предикат, который требуется использовать чаще всего, например, сравнение на равенство. Текст нового предиката с двумя аргументами приведен в листинге 12.9.

Листинг 12.9. Универсальный функтор-предикат

```
template <class type_elem, typename Comp = equalsto<type_elem> >
class Compares
{  public:
   Compares(const type_elem& val) : v_(val) {}
   bool operator() ( type_elem val) { return Comp(val); }
   private:
   type_elem v_;
};
```

Тогда прототип фильтра с новым параметром-предикатом выглядит так:

```
template <class T, class Comp >
vector<Person> filter(const vector<Person> &vv, Compares<T, Comp>(x))
```

Вызов по умолчанию будет таким:

```
vector<Person> bb(10);
bb = filter(bb, Compares<int>(3));
```

Вызов с явно заданным предикатом выглядит так:

```
bb = filter(bb, Compares<Date, lessthen<Date> >(Date(22, 02, 1953)));
```

Решения стандартной библиотеки

В стандартной библиотеке функции-объекты играют большую роль, поэтому разработчики очень тщательно подошли к разработке и реализовали их с максимальной общностью. В STL определены два класса: `unary_function` и `binary_function`. Эти классы "ничего не делают", предоставляя просто имена для аргументов и результата. Все функциональные объекты библиотеки наследуют от этих классов. В листинге 12.10 приведена структура этих классов и примеры наследования из стандартной библиотеки.

Листинг 12.10. Иерархия функций-объектов в стандартной библиотеке

```
#include <functional>
template <class Arg, class Result>
struct unary_function {
typedef Arg argument_type;
typedef Result result_type;
};
template <class Arg1, class Arg2, class Result>
struct binary_function {
typedef Arg1 first_argument_type;
typedef Arg2 second_argument_type;
typedef Result result_type;
};
template <class _Tp>
struct equal_to : public binary_function<_Tp,_Tp, bool>
{ bool operator()(const _Tp& x, const _Tp& y) const { return x == y; }
};
template <class _Tp>
struct plus : public binary_function<_Tp,_Tp,_Tp>
{ _Tp operator()(const _Tp& x, const _Tp& y) const { return x + y; }
};
```

```
template <class _Tp>
struct negate : public unary_function<_Tp,_Tp>
{_Tp operator()(const _Tp& x) const { return -x; }
};
template <class _Tp>
struct logical_and : public binary_function<_Tp,_Tp,bool>
{ bool operator()(const _Tp& x, const _Tp& y) const { return x && y; }
};
template <class _Tp>
struct logical_not : public unary_function<_Tp, bool>
{ bool operator()(const _Tp& __x) const { return !__x; }
};
```

Обратите внимание, что от одного и того же класса наследуют функции совершенно разной природы: и арифметические, и логические. Такая организация позволяет максимально обобщить алгоритмы, указав в определении формальный параметр базового класса, а фактически передавать объект производного класса (открытое наследование в действии). Мы и сами можем создать предикат, унаследовав от базового класса. В справочной системе Borland C++ Builder представлен пример такого наследования, который приведен в листинге 12.11 в упрощенном виде.

Листинг 12.11. Наследование от функции-объекта стандартной библиотеки

```
template<class Arg>
class factorial : public unary_function<Arg, Arg>
{  public:
   Arg operator()(const Arg& arg)
   {  Arg a = 1;
      for (Arg i = 2; i <= arg; i++) a *= i;
      return a;
   }
};
```

Это функция-объект, вычисляющая факториал своего аргумента. Пользоваться объектом можно, например, так:

```
int init[7] = { 1, 2, 3, 4, 5, 6, 7 };
deque<int> d(init, init + 7);    //  создали контейнер-дек
vector<int> v(7);                //  создали пустой контейнер-вектор
transform(d.begin(), d.end(), v.begin(), factorial<int>());
```

Алгоритм `transform` занесет в вектор факториалы значений дека.

В стандартной библиотеке реализованы два вида объектов-адаптеров для функций-объектов: *отрицатели* и *привязки* (связыватели). Как можно заме-

тить, предикаты сравнения в стандартной библиотеке реализованы в максимально общем виде — с двумя аргументами общего вида. Привязка — это объект-функция, который превращает бинарный предикат в унарный, фиксируя один из аргументов. Так как аргументов у бинарного предиката два, то и привязок в библиотеке тоже две. Например, мы хотим сравнивать с числом 100 значение элемента контейнера-вектора `v` из предыдущего примера (см. листинг 12.10) в алгоритме `find_if`. Тогда на месте параметра-предиката нужно написать следующее:

```
bind2nd(equal_to<int>, 100)
```

Этим мы зафиксировали значение второго элемента и превратили бинарный предикат в унарный.

Отрицатель — это функциональный объект, аналогичный по действию оператору логического отрицания `!`. Можно было бы перегрузить операцию логического отрицания, но тогда это надо делать в каждом функциональном классе. Разработчики стандартной библиотеки пошли по другому пути: реализовали универсальные функциональные объекты для унарных и бинарных предикатов. Если нам нужно использовать в алгоритме поиска неравенство вида:

```
! (x < 100)
```

то в функциональной форме мы должны написать:

```
not1(bind2nd(less<int>, 100))
```

Отрицатель для бинарных предикатов имеет имя `not2`.

Объектно-ориентированное обобщенное программирование

Как уже было сказано, объектно-ориентированное обобщенное программирование — это программирование паттернов [10] и реализация обобщенных структур данных — контейнеров. С контейнерами мы уже неоднократно сталкивались и даже реализовали шаблон контейнера-дека *(см. гл. 9)*, поэтому сейчас выясним, что же такое *паттерны проектирования*. "Библией" в данном вопросе считается книга [10], в которой описано 23 паттерна. Паттерны проектирования — это приемы разработки программного кода, способствующие повышению качества программного кода и облегчающие последующие изменения. В объектно-ориентированном программировании это означает определенным образом организованное взаимодействие классов и объектов.

Рассмотрим такой простой паттерн, как `Singleton` ("одиночка"). "Одиночка" — это объект, который существует в программе в единственном эк-

зкземпляре, и клиент никаким образом не может создать второй такой же объект. Такие объекты постоянно встречаются в программировании. Например, при моделировании компьютера нужно гарантировать единственность процессора или клавиатуры. При программировании для Windows часто требуется контролировать единственность исполняемого экземпляра задачи. Каждый элемент множества тоже должен быть единственным.

Так как на "честность" клиента полагаться нельзя, то сам класс должен обеспечивать такой контроль. Первое, что надо сделать — закрыть конструктор, а вместо него предоставить функцию, которая и будет создавать объект. Второе — эта же функция будет считать количество объектов и предотвращать создание новых. Здесь надо использовать статические элементы класса. Реализация простейшего варианта "одиночки" приведена в листинге 12.12.

Листинг 12.12. Реализация паттерна `Singleton`

```
class Singleton
{  public:
   static Singleton * Instance();    //  функция создания объекта
   private:
   Singleton();    //  запрет непосредственного создания объекта
   Singleton(const Singleton &sg);
   Singleton& operator=(const Singleton &sg);    //  закрыто присваивание
   ~Singleton();    //  закрыт конструктор
   static Singleton * instance_;
};
Singleton * Singleton::instance_= 0;    //  обязательная инициализация
Singleton * Singleton::Instance()
{  if (instance_ == 0) instance_ = new Sigleton();
   return instance_;
};
```

В открытой части объявляются необходимые функции-методы. Пользователь должен в программе объявить указатель на объект типа `Singleton` и присвоить ему значение, вызвав функцию-создателя:

```
Singleton *p = Instance();
```

Доступ к открытым функциям-методам выполняется косвенно через указатель.

Одним из самых простых и понятных паттернов является паттерн `Adapter`, который мы уже неоднократно упоминали. В стандартной библиотеке шаблонов реализовано несколько адаптеров контейнеров и итераторов.

Смысл этого паттерна заключается в том, что один класс адаптирует интерфейс другого класса к конкретным нуждам клиента, скрывая то, без чего клиент может обойтись. Рассмотрим реализацию этого паттерна на конкретном примере. В *гл. 9* мы реализовали шаблон контейнера-дека. Однако клиентам не нужна излишняя общность дека — предпочтение отдается частным случаям дека: стеку и очереди. Поэтому мы можем реализовать адаптер, ограничив для клиента интерфейс дека.

При реализации паттерна требуется использовать множественное наследование. Открытым образом адаптер наследует от абстрактного класса-интерфейса, который виден клиенту, а от исходного класса наследование закрытое. На языке C++ это выглядит так:

```
class Adapter: public Queue, private Deque
{ //... };
```

Абстрактные классы

Абстрактным классом называется класс, в котором объявлена (или унаследована) хотя бы одна "чистая" виртуальная функция.

Виртуальная функция является "чисто виртуальной", если она не имеет тела. Отсутствие тела задается присвоением нуля, например:

```
virtual void f(void) = 0;
```

Если в классе объявлена хотя бы одна такая функция, то класс считается абстрактным, и объекты такого класса объявлять нельзя. Указатели и ссылки на абстрактный класс объявлять допускается, если при инициализации не требуется создавать временный объект. Наследовать от абстрактного класса разрешается, но наследник сам будет считаться абстрактным классом, если не определит реализацию унаследованной "чисто виртуальной" функции. Деструктор не может быть "чисто виртуальным", поскольку деструкторы производных классов всегда вызывают деструктор базового класса.

Абстрактные классы чрезвычайно полезны. Обычно абстрактный класс является вершиной иерархии, в котором собраны наиболее общие свойства классов иерархии. Например, хорошим кандидатом в абстрактный класс является уже упомянутый нами класс `Shape` (Форма), от которого наследуют конкретные классы-фигуры. Другим примером может служить абстрактный класс `Window` (Окно), наследниками которого могут быть разные виды конкретных окон в диалоговой системе.

С помощью абстрактного класса очень удобно представлять интерфейс. Обычно под интерфейсом понимают абстрактный класс, включающий только "чистые" виртуальные функции и константы. На таком понятии интерфейса основана вся компонентная технология [4], реализованная в рамках операционной системы Windows.

Абстрактный класс позволяет отделить интерфейс от реализации, тем самым, повышая инкапсуляцию проекта. Это позволяет развивать реализацию практически независимо от интерфейса, предоставляемого клиенту.

Абстрактный класс-интерфейс очереди приведен в листинге 12.13.

Листинг 12.13. Интерфейс очереди

```
template <class T>
class Queue
{  public:
   class EmptyQuee{};     //  исключение
   explicit Queue();      //  конструктор
   bool empty() const = 0;
   const T front() const = 0;
   const T back() const = 0;
   void push_back(const T &a) = 0;
   void pop_front() = 0;
};
```

В абстрактном классе обычно отсутствуют поля данных. Класс-адаптер должен реализовать все виртуальные функции, пользуясь методами адаптируемого класса.

Метапрограммирование

В 1994 году, во время встречи комитета по стандартизации, Эрвин Анрух (Erwin Unruh) обнаружил удивительное свойство шаблонов — способность выполнять реальные вычисления. Фокус заключается в том, что эти вычисления выполняются во время трансляции, а не во время выполнения. Поначалу именно эта способность шаблонов получила название метапрограммирование, но потом этот термин стали применять в более общем смысле — метапрограммирование шаблонов.

Продемонстрируем эту интереснейшую способность шаблонов: напишем программу, возводящую 2 в целую положительную степень. Но сначала, чтобы разобраться в существе дела, создадим простую функцию, которая делает то же самое:

```
double pow2(const int n)
{  double n2 = 1;
   for (int i = 1; i < n+1; ++i) n2 *= 2;
   return n2;
}
```

Функция вызывается так:

```
cout << pow2(0) << endl;
cout << pow2(10) << endl;
```

Вызов функции происходит во время выполнения программы. Это значит, что в исполняемом модуле присутствуют команды, которые реально выполняет процессор.

Однако надо заметить, что в данном случае параметром является константа. Мы знаем, что константные выражения компилятор может вычислить на этапе трансляции, поэтому такие вызовы функций во время выполнения программы, вообще-то говоря, излишни. Именно при вычислениях константных выражений могут применяться шаблоны. Метапрограммирование принципиально нельзя использовать, когда параметром является переменная или произвольное выражение — запомните это.

Еще один принципиальный момент требует разъяснения: написание шаблона функции — это совсем не то, что нам требуется. Когда компилятор "видит" шаблон функции, он проверяет его синтаксис, и игнорирует до тех пор, пока не "увидит" вызов. Тогда компилятор на основании уже собранной информации о типах определяет типы параметров шаблона, "возвращается" к определению шаблона функции и генерирует реальное определение функции, подставляя вместо параметров шаблона соответствующие типы. Это определение попадает в исполняемый модуль, и вызовы работают во время выполнения программы. Таким образом, шаблон фактически является макросом, правильность записи которого проверяет компилятор. Нам же требуется совсем другое: выполнить вычисление степени двойки во время компиляции и подставить это значение на место вызова. Тогда при выполнении программы оператор вывода будет выводить константу. Понятно, что вывод константы выполняется значительно быстрее, чем вывод значения, вычисляемого функцией.

Однако написать программу, выполняющую метавычисление не так-то просто. Вдумайтесь: требуется выполнить оператор цикла во время компиляции. В языке C++ нет такого оператора. Более того, в языке вообще отсутствуют какие-то ни было средства периода компиляции (препроцессор не является частью компилятора — это отдельная программа). Тем не менее вычисления на этапе компиляции возможны — вместо цикла нужно использовать рекурсию. Вспомним рекурсивное определение степени двойки:

- $2^0 = 1$
- $2^n = 2 \times 2^{n-1}$

Первое выражение — условие окончания вычислений, второе — рекурсивное определение. Следовательно, если мы сможем написать два взаимосвязанных шаблона, соответствующие этому определению, то "дело в шляпе". И это можно сделать — текст программы приведен в листинге 12.14.

Листинг 12.14. Метавычисление степени двойки

```
#ifndef POW2_HPP
#define POW2_HPP
  //  рекурсивное вычисление 2 в степени N
template<int N>
class Pow2 {
public: enum { result = 2 * Pow2<N-1>::result };
};
  //  условие окончания рекурсии — полная специализация шаблона
template<>
class Pow2<0> {
public: enum { result = 1 };
};
#endif    //  POW2_HPP
```

Мы оформили два шаблона в отдельный включаемый файл, чтобы его можно было подключать по мере необходимости. Например, так:

```
#include <iostream>
#include "pow2.hpp"    //  подключение шаблонов
int main()
{ std::cout << "Pow2<10>::result = " << Pow2<10>::result << '\n'; }
```

Поясним несколько деталей. Основной шаблон соответствует второму выражению определения степени. Чтобы завершить вычисления, используется полная специализация шаблона. Вычисления, естественно, целочисленные, и параметр шаблона — целый.

К сожалению, на данном этапе развития C++ в качестве непосредственных аргументов шаблона не могут использоваться дробные типы. Кроме того, надо помнить об ограничениях вычислений времени компиляции. Например, данный шаблон прекрасно вычисляет все степени до тридцатой включительно, однако при вычислении 2^{31} происходит переполнение и результат получается отрицательным. Мы уже сталкивались с подобным явлением *(см. гл. 1)*.

Совершенно аналогично можно вычислять факториал (с учетом указанных ограничений при вычислениях). Кроме того, заменим класс на структуру, а перечислимый тип — объявлением статической константы:

```
template<int n>
struct Factorial {
static int const value = n * Factorial<n-1>::value;
};
```

```
template<>
struct Factorial<0> {
static int const result value = 1;
};
```

Примечание

Этот пример работает только в системе Borland C++ Builder 6, т. к. статические константы в остальных системах, используемых в данной книге, не реализованы.

Если вместо полной специализации использовать частичную, то можно вычислять более сложные выражения, например, K^n (текст программы приведен в листинге 12.15).

Листинг 12.15. Вычисление K^n

```
template<int K, int N>
class Pow {
public: enum { result = K * Pow<K, N-1>::result };
};
  //  частичная специализация шаблона
template<int K>
class Pow<K, 0> {
public: enum { result = 1 };
};
```

"Вызов" (инстанцирование) шаблона делается так:

```
cout << "Pow<3, 0>::result = " << Pow<3, 0>::result << '\n';
cout << "Pow<5, 3>::result = " << Pow<5, 3>::result << '\n';
```

Рассмотрим еще более сложный пример: реализуем скалярное произведение двух обычных массивов. Традиционно программисты пишут шаблон такого вида:

```
template <typename T>
inline T product (int dim, T* a, T* b)
{  T result = 0;
   for (int i = 0; i < dim; ++i)
   result += a[i] * b[i];
   return result;
}
```

Инстанцирование шаблона в конкретную функцию и ее вызов выполняется, например, в следующем контексте:

```
int main()
{  double a[3] = { 1, 2, 3 };
   double b[3] = { 5, 6, 7 };
   cout << "product(3, a, b) = " << product(3, a, b) << endl;
   cout << "product(3, a, a) = " << product(3, a, a) << endl;
}
```

Используя шаблоны, можно избавиться от цикла внутри функции — на месте цикла будет инстанцировано непосредственно выражение скалярного произведения. Например, для первого вызова приведенного примера это будет выражение:

```
a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
```

Текст программы представлен в листинге 12.16.

Листинг 12.16. Обобщенное вычисление скалярного произведения

```
  //  первичный шаблон
template <int DIM, typename T>
class Product {
public:
static T result (T* a, T* b) {
return *a * *b + Product<DIM-1, T>::result(a+1, b+1);
  }
};
  //  частичная специализация для окончания рекурсии
template <typename T>
class DotProduct<1, T> {
public:
static T result (T* a, T* b) { return *a * *b; }
};
  //  шаблон функции — "обертка" вокруг шаблона Product
template <int DIM, typename T>
inline T product (T* a, T* b)
{ return Product<DIM, T>::result(a, b); }
```

Обратите внимание, что в первичном шаблоне объявлена статическая рекурсивная функция. Шаблон функции-"обертки" необходим только для того, чтобы вызвать функцию первичного шаблона первый раз. Вызов функции-"обертки" практически не отличается от вызова функции с циклом:

```
cout << "product<3>(a, b) = " << product<3>(a, b) << endl;
cout << "product<3>(a, a) = " << product<3>(a, a) << endl;
```

На этом закончим экскурс в обобщенное программирование, хотя тема эта практически неисчерпаема.

ЧАСТЬ III

Как написать программу для Windows

Глава 13. Windows — это не просто

Глава 14. Быстрая разработка приложений

ГЛАВА 13

Windows — это не просто

Любая операционная система в своем составе обязательно имеет довольно большой набор функций, которые можно использовать в прикладных программах. Каждая система программирования обеспечивает языковой интерфейс этих функций. Мы уже пользовались функцией перекодировки `CharToOem`, функциями для проверки и установки текущего каталога. С "легкой руки" Microsoft набор таких функций стали называть (по-англ.) *Application Programming Interface*. По-русски это можно перевести как "Интерфейс прикладного программирования", однако, как это часто бывает в программировании, русским термином никто не пользуется. Общепринятое сокращение такого длинного названия — *API*. API Windows сокращенно называют *WinAPI* или *Win32 API*. Число 32 указывается для того, чтобы подчеркнуть отличие нынешней, 32-разрядной версии от прежней 16-разрядной. Уже существует следующая версия, которая называется Win64 [33, 43].

WinAPI включает не только более 2000 функций, но и огромное количество констант, типов и структур данных. Изучить все это "хозяйство" достаточно сложно даже для опытных программистов. Наша задача — хотя бы немного разобраться в основах организации прикладных программ, которые работают в Windows. Начнем с самого простого — обозначений.

Венгерская нотация

Так как операционная система — очень большая программа, она создается группой программистов. Очень важно, чтобы каждый программист мог читать и понимать обозначения констант, переменных, типов, функций другого программиста — это существенно облегчает коллективную работу. Программисты Microsoft разработали собственную систему обозначений, которую применяют теперь все программисты, создающие программы для Windows.

Если мы "зайдем" в каталог include любой интегрированной среды, то увидим там многочисленные файлы, название которых начинается с префикса win. В этих файлах содержится огромное количество определений констант, типов, структур данных. Программисты Microsoft использовали для этого оператор `typedef` или директиву препроцессора `#define`. Такое широкое использование устаревших по современным понятиям средств обусловлено историческими причинами. Операционная система Windows создавалась задолго до принятия стандартов C и C++. Более того, первые пробные версии писались на Ассемблере и Pascal, но потом ее переписали на C. В последних версиях некоторые части написаны уже на C++, но никто не переписывал то, что уже было написано на C. На обозначения повлияла и старая сегментная архитектура компьютера IBM PC, для которого изначально создавалась Windows.

Даже обычное нежелание программистов писать длинные имена сыграло свою роль. Поэтому, например, в файле windef.h можно найти такие определения типов:

```
typedef unsigned char UCHAR;
typedef int INT;
typedef int BOOL;
typedef unsigned short USHORT;
typedef unsigned int UINT;
typedef long LONG;
typedef unsigned long ULONG;
typedef unsigned long DWORD;
typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef float FLOAT;
```

Как видите, все сделано в единообразном стиле. Однако группа разработчиков Windows пошла дальше: Чарльз Симони предложил нотацию, которая в его честь стала в дальнейшем называться венгерской. Все очень просто: имя переменной (константы) начинается с буквы или букв, которые обозначают тип этой переменной (константы). Например, префикс sz означает, что строка завершается нулем (`string terminated by zero`); префикс i означает целое (`integer`); префикс p (`pointer`) обозначает указатель. Тот же принцип часто применяется и при обозначении типов. Например, в том же файле можно увидеть такие определения типов указателей:

```
typedef FLOAT *PFLOAT;
typedef ULONG *PULONG;
typedef USHORT *PUSHORT;
typedef UCHAR *PUCHAR;
typedef char *PSZ;
```

Мы уже встречались с именами указателей, которые начинаются на букву L (вспомним функцию перекодировки `CharToOem` — см. листинг 4.8). Не будем вдаваться в исторические причины появления этой буквы, скажем лишь, что буква L перед именем означает long — "длинный". В Win32 API все указатели "длинные".

Во включаемых файлах с префиксом win часто можно встретить переопределения типов:

```
typedef UINT WPARAM;
typedef LONG LPARAM;
typedef LONG LRESULT;
```

Это сделано исключительно для удобства чтения программ.

Таким образом, имена стандартных типов данных, используемых в API, нужно писать прописными буквами. Обычно большими буквами записываются и константы, определенные с помощью директивы `#define`.

Венгерская нотация помогает избегать ошибок в программе, состоящей из многих частей, еще до компоновки. Поскольку имя переменной (константы) описывает и саму переменную, и тип ее данных, то намного снижается вероятность введения в программу ошибок, связанных с несовпадением типа данных у переменных. Некоторые наиболее часто используемые префиксы приведены в табл. 13.1.

Таблица 13.1. *Префиксы в венгерской нотации*

Префикс	Тип данных
c	Символ (`char`); иногда — константа (`const`)
by	Беззнаковый символ (`byte`)
n	Короткое целое
i	Целое
x, y	Целое (используется в качестве координат x и y)
cx, cy	Целое, "c" означает "счет" — (`count`)
b или f	Булево целое (`bool`); "f" означает "флаг" — (`flag`)
w	Беззнаковое короткое целое (`word`)
l	Длинное целое (`long`)
u	Беззнаковое (`unsigned`)
dw	Беззнаковое длинное целое (`dword`)
fn	Функция

Таблица 13.1 *(окончание)*

Префикс	Тип данных
s	Строка
sz	Строка, завершаемая нулем
h	Описатель (`handle`)
p	Указатель (`pointer`)

Таким образом, имена получаются "самодокументированными". Например, имя `lpszFileName` — это, очевидно, длинный указатель на строку символов, которая представляет собой имя файла; `fdwAccess` — двойное слово, содержащее флаги, определяющие доступ (к чему-нибудь).

Отдельно надо сказать о префиксе h, который произошел от английского слова handle (ручка, рукоятка). В таблице это слово переведено как "описатель". Иногда этот описатель называют дескриптором [43] (по англ. descriptor). И тот, и другой термин не совсем соответствуют действительности, поэтому мы будем пользоваться калькой "хендл". На самом деле "хендл" не является ни описателем, ни дескриптором — это просто целое беззнаковое число, которое генерирует система для каждой сущности в программе. Каждое окно, каждый файл, каждый ресурс имеет свой уникальный "хендл". Более того, сама работающая в Windows программа имеет свой "хендл". Мы постоянно будем сталкиваться с этим понятием при программировании для Windows.

Консольные приложения и Unicode

Как известно, множество целых чисел, которые представляют символы в памяти компьютера, называется *кодировкой* или *кодом символов*. До сих пор в наших консольных программах использовался стандартный американский код, который в документации Microsoft называется ASCII. Для обозначения символьных переменных и констант в программах использовался стандартный символьный тип `char`. Размер этого типа в C++ равен единице (на платформе Intel это 1 байт). При написании программы в интегрированной среде, предназначенной для разработки программ для Windows, приходилось вызывать функцию перекодировки, т. к. Windows использует другой код для символов.

Программистское сообщество давно было озабочено проблемами, связанными с разными кодировками символов. Для решения проблем перекодирования была разработана универсальная кодировка, которая по-английски так и называется Unicode. На платформе Intel символ в кодировке Unicode занимает 2 байта. В язык C был добавлен специальный тип `wchar_t`, который предназначался для представления "расширенных" символов.

Примечание

В справочной системе Borland C++ Builder 6 написано, что тип `wchar_t` имеет тот же размер, что и тип `int`. Это вызывает недоумение, поскольку "расширенные" символы обычно занимают два байта, а целые числа — четыре.

Константы, состоящие из "расширенных" символов, объявляются со специальным префиксом L:

```
wchar_t *ws = L"Русская строка!";
```

Для работы с такими строками в C был определен полный набор функций с префиксом wcs, аналогичных соответствующим функциям с префиксом str. Сравните два прототипа:

```
char *strcat(char *dest, const char *src);
wchar_t *wcscat(wchar_t *dest, const wchar_t *src);
```

Как видим, прототипы во всем идентичны, кроме типа символов. Однако стандартные потоки не работают с такими символами, поэтому оператор

```
cout << ws << endl;
```

выведет на экран совсем не строку, а значение указателя (адрес строки).

Для работы со строками в Windows надо пользоваться средствами WinAPI. WinAPI поддерживает стандартные однобайтовые символы (тип `char` или `CHAR`) и "расширенные" двухбайтовые (тип `WCHAR`, определенный в файле wtypes.h). Однако не все версии Windows "умеют" работать с Unicode. Именно поэтому Microsoft реализовала в составе WinAPI универсальный тип `TCHAR` (определенный в tchar.h), с помощью которого можно писать универсальные программы, которые, в зависимости от версии системы, будут использовать либо обычные, либо расширенные символы. При этом перекодировка не потребуется.

Рассмотрим подробнее создание консольных приложений, использующих универсальные символы. Снова напишем программу вывода таблицы умножения. Как обычно, программа должна выводить приглашение задать множитель, ввести его, вычислить и вывести в столбик таблицу умножения.

Чтобы все корректно работало, нужно придерживаться нескольких простых правил.

1. Все символьные строки объявляются с помощью типов `TCHAR`, `LPTSTR`, `LPCTSTR`.
2. Строковые константы должны иметь одну из трех форм (это правило касается и отдельных символов):
   - `_T("Универсальные символы");`
   - `_TEXT("Универсальные символы");`
   - `TEXT("Универсальные символы").`

3. Длина символьных буферов должна вычисляться с помощью операции `sizeof(TCHAR)`.
4. После `<windows.h>` нужно включить `<tchar.h>`. Этот заголовок содержит необходимые определения для макросов и универсальных функций библиотеки C.
5. Перед заголовком `<windows.h>` необходимо включить два следующих определения: `#define UNICODE` и `#define _UNICODE`.

Для вывода на консоль в программе необходимо выполнить несколько действий:

1. Открыть файл на стандартном устройстве вывода.
2. Установить для файла консольный режим.
3. Вывести столько сообщений, сколько необходимо.
4. Закрыть файл.

Текст программы, вычисляющей таблицу умножения, приведен в листинге 13.1.

Листинг 13.1. Консольная программа с использованием Unicode

```
#define UNICODE          //  определение Unicode
#define _UNICODE
#include <windows.h>
#include <tchar.h>       //  определение макросов
#include <conio.h>
#include <iostream.h>    //  для ввода множителя
  //  наши функции вывода на консоль
BOOL PrintMsg(HANDLE hOut, LPCTSTR pMsg);
BOOL PrintStrings(HANDLE hOut, ...);

int main(char argc, char ** _targv)
{  int k;                    //  множитель
   HANDLE hStdOut;           //  "хендл" консоли
     //  открываем файл на стандартном устройстве вывода
   hStdOut = CreateFile(_T("CONOUT$"), GENERIC_WRITE, 0, NULL,
   OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
   SetConsoleMode(hStdOut,    //  установили режим консоли
   ENABLE_WRAP_AT_EOL_OUTPUT | ENABLE_PROCESSED_OUTPUT);
   TCHAR* pr = _T("Введите множитель: ");    //  текст приглашения
   PrintMsg(hStdOut, pr);                    //  вывод приглашения
   cin >> k;                                 //  ввод множителя
```

```
    //  вывод заголовка таблицы — большими буквами
  PrintMsg(hStdOut, CharUpper(_T("Таблица умножения:\n")));
    // перевод множителя в строку
  TCHAR str1[3]; _itot(k, str1, 10); str1[2] = 0;
  for (int i = 0; i <= 10; i++)  //  цикл вычисления и перевода в строку
  {  TCHAR str2[3]; _itot(i, str2, 10); str2[2] = 0;
     int d = k * i;                  //  вычисляем результат
     TCHAR str3[3]; _itot(d, str3, 10); str3[2] = 0;
     TCHAR *st = new TCHAR[25];      //  объединенная строка
     for (int j = 0; j < 25; ++j)    //  обнуляем строку
     st[i] = _T("\0");
       //  формируем строку для вывода
     st = wcscat(st, str2);
     st = wcscat(st, _T("*"));
     st = wcscat(st, str1);
     st = wcscat(st, _T("="));
     st = wcscat(st, str3);
     st = wcscat(st, _T("\n"));
     PrintMsg(hStdOut, st);          //  выводим строку
  }
  getch();                           //  остановить и посмотреть
  return 0;
}
```

В программе объявляется переменная-"хендл", которой присваивается значение при открытии файла. В Windows два имени файла CONIN$ и CONOUT$ зарезервированы для консольного ввода и вывода. В нашем случае мы открываем файл для вывода. Остальные параметры, представленные константами Windows, определяют режимы открытия файла.

Далее мы используем "хендл" открытого файла для установки режима работы консоли, который задается соответствующими константами. После этого просто выводится приглашение, вводится множитель и выводится заголовок таблицы умножения. В данном случае продемонстрировано использование функции перевода в верхний регистр. Обратите внимание, что функции вывода передается в качестве параметра "хендл" открытого файла.

Основная часть программы — это цикл, в котором формируется очередная строка таблицы умножения. Сначала множители и результат переводятся в символьную форму, а затем, с помощью функции сцепления, строка формируется целиком. Используются функции, обрабатывающие универсальные символы.

Теперь рассмотрим функции вывода, текст которых приведен в листинге 13.2.

Листинг 13.2. Функции вывода сообщений на консоль

```
BOOL PrintStrings(HANDLE hOut, ...)
{  DWORD MsgLen, Count;
   LPCTSTR pMsg;
   va_list pMsgList;
   va_start(pMsgList,hOut);
   while ((pMsg = va_arg(pMsgList, LPCTSTR)) != NULL)
   {  MsgLen = lstrlen(pMsg);
      if (!WriteConsole(hOut, pMsg, MsgLen, &Count, NULL)) return FALSE;
   }
   return TRUE;
};
BOOL PrintMsg(HANDLE hOut, LPCTSTR pMsg)
{ return PrintStrings(hOut, pMsg, NULL); }
```

Функция `PrintString` написана как функция с переменным числом параметров, с использованием стандартных макросов `va_...` Собственно вывод на консоль выполняется функцией `WriteConcole`, которая в качестве параметра как раз получает "хендл" файла, выводимую строку и ее длину, которая вычисляется функцией WinAPI, работающей с универсальными символами.

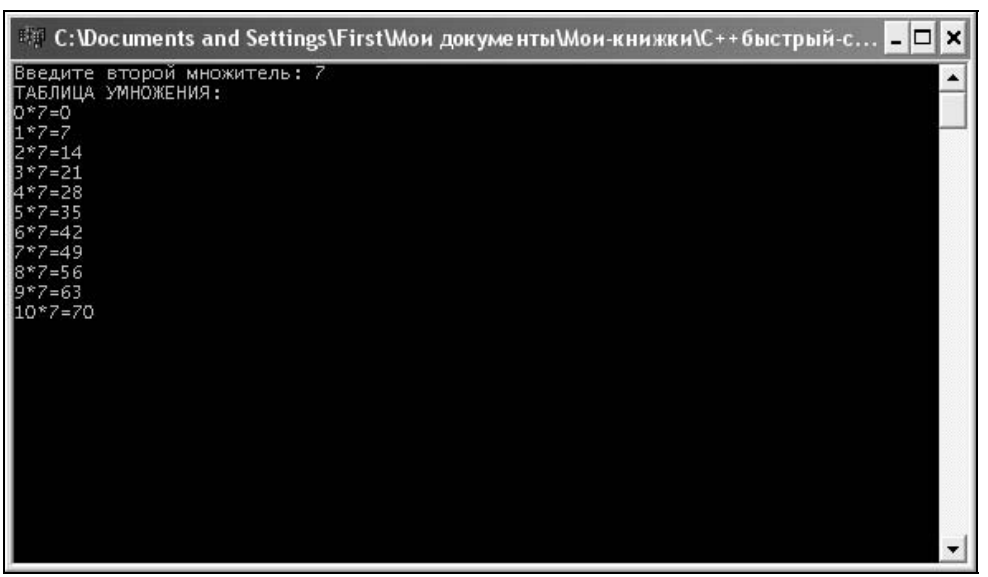

Рис. 13.1. Вывод программы вычисления таблицы умножения

Вторая функция просто использует первую с единственным аргументом в списке переменной длины.

Результат работы нашей программы представлен на рис. 13.1.

Набор консольных функций достаточно велик, поэтому в небольшой главе привести их нет никакой возможности. Обо всех них можно прочитать в справочной системе Borland C++ Builder 6 в разделе MS SDK Help Files или в справочной системе Microsoft MSDN, которая доступна в Интернете.

Оконные приложения

Операционная система Windows получила такое название именно потому, что основным видом приложений в ней являются "оконные" программы. Для пользователя на экране дисплея все выглядит замечательно красиво, и, что самое главное, — единообразно. Однако простота для пользователя частенько оборачивается "головной болью" программиста. Тем не менее попробуем разобраться в принципах организации оконных программ — реализуем оконный вариант нашей программы "Таблица умножения", используя только WinAPI. Поскольку работа довольно сложная, будем двигаться постепенно. Сначала создадим "пустое" приложение, а потом будем "навешивать" на него необходимую нам функциональность. Это типичная процедура при программировании оконных приложений для Windows.

Создание "пустой" программы

Оконная программа имеет достаточно "жесткую" структуру и состоит из двух основных функций: главной и функции окна, которую часто называют *оконной процедурой*. Вся основная работа выполняется в функции окна. Кроме этих обязательных функций программист может писать любое количество других функций, которые требуются для решения задачи.

Главная функция — это отнюдь не `main`, как мы привыкли. Главная функция оконного приложения называется `WinMain`, и в ней необходимо соблюсти строго определенную последовательность действий.

1. Зарегистрировать класс окна.
2. Создать окно.
3. Отобразить окно.
4. Выполнить цикл обработки сообщений.

Иногда (но редко) выполняются некоторые другие действия, но мы пока на этом останавливаться не будем. Текст типичной главной функции приведен в листинге 13.3. Эта функция фактически является шаблоном, на основе которого создаются главные функции в других оконных программах.

Листинг 13.3. Главная функция оконного приложения

```
int WINAPI WinMain( HINSTANCE hI,    //  "хендл" приложения (текущая)
                    HINSTANCE hP,    //  "хендл" приложения (предыдущая)
                    LPSTR lpszCmdLine,    //  параметры командной строки
                    int nCmdLine)    //  вид на экране
{  MSG msg;      //  сообщение
   HWND hwnd;    //  "хендл" окна
   static char szClassName[] = "FirstProgram";    //  класс окна
     //  регистрация класса окна
   if (!RegClass(hI, WndProc, szClassName)) return FALSE;
     //  создание главного окна
   hwnd = CreateWindow(
     szClassName,                        //  имя класса окна
     "Таблица умножения",                //  заголовок окна
     WS_OVERLAPPEDWINDOW|WS_VISIBLE,     //  стиль окна
     CW_USEDEFAULT, CW_USEDEFAULT,       //  координаты окна
     CW_USEDEFAULT, CW_USEDEFAULT,       //  размеры окна
     0,        //  родителя нет
     0,        //  меню нет
     hI,       //  наш "хендл"
     NULL);    //  дополнительно
   if (!hwnd) return FALSE;    //  если не создалось — выходим
   ShowWindow(hwnd, SW_SHOWNORMAL);    //  показать окно
   UpdateWindow(hwnd);                 //  перерисовать окно
   while(GetMessage(&msg, 0, 0, 0))    //  цикл обработки сообщений
   {  TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
   return msg.wParam;
}
```

В главной функции описано четыре параметра, однако фактически используется только первый из них: операционная система помещает в этот параметр "хендл" приложения.

В данном случае для реальной работы нам не хватает только двух функций: регистрации класса окна с именем `RegClass` и оконной процедуры (функции окна). Функция регистрации (ее текст приведен в листинге 13.4) — это наша функция, написанная специально для того, чтобы инкапсулировать все, связанное с регистрацией класса окна. Внутри функции в переменной класса окна инкапсулируется вся информация, общая для данного типа окон, а при создании конкретного окна задаются определенные параметры окна данного типа.

Листинг 13.4. Функция регистрации класса окна

```
BOOL RegClass(HINSTANCE hInst, WNDPROC WndProc, LPCTSTR szName)
{  WNDCLASS wc;    //  переменная класса окна
   wc.style = CS_HREDRAW | CS_VREDRAW;     //  стиль окна
   wc.lpfnWndProc = WndProc;               //  А, функция окна
   wc.lpszMenuName = NULL;                 //  Б, меню нет
   wc.lpszClassName = szName;              //  фиксируем класс окна
   wc. hInstance = hInst;                  //  "хендл" приложения
     //  загрузка ресурсов
   wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
   wc.hCursor = LoadCursor(NULL, IDC_ARROW);
     // закраска рабочей области
   wc.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH); // класс окна
   wc.cbClsExtra = wc.cbWndExtra = 0;
   return (RegisterClass(&wc) != 0);       //  регистрация
}
```

Самыми важными являются несколько первых строчек. Оператор

```
wc.style = CS_HREDRAW | CS_VREDRAW;
```

устанавливает, что все окна данного класса должны перерисовываться при изменении горизонтального (`CS_HREDRAW`) и вертикального (`CS_VREDRAW`) размеров окна. Это важно, если в окно выполняется вывод информации с выравниванием относительно границ окна.

Строка `А` фиксирует, какой функцией окна будут обрабатываться сообщения. В данном случае имя оконной процедуры должно быть `WndProc`.

В строке `Б` устанавливается, что меню отсутствует (потом мы его добавим).

Операторы

```
wc.lpszClassName = szName;     //  фиксируем класс окна
wc.hInstance = hInst;          //  "хендл" приложения
```

фиксируют класс окна и связывают данный класс с нашим приложением. Класс окна — это просто строка символов. Желательно, чтобы эта строка была уникальной, т. к. по ней можно будет идентифицировать все окна данного приложения.

Далее выполняется загрузка ресурсов (пиктограмма и курсор). В данном случае, поскольку первый параметр нулевой, никакой загрузки не происходит. А вообще-то говоря, мы можем создать пиктограмму (файл с расширением ico) и курсор (файл с расширением cur) и подключить их к своей программе. Но работа с ресурсами — отдельная большая тема, поэтому здесь мы этим заниматься не будем.

Закраска рабочей области выполняется стандартной кистью темно-серого цвета.

Наконец, в последней строке, происходит регистрация класса окна — вызывается функция API `RegisterClass`. Если по каким-то причинам регистрация закончится неуспешно, то в главную функцию попадет ненулевой результат, и наше приложение завершит работу.

Вернемся к главной функции. Функция создания окна выдает "хендл" окна, если оно успешно создано. При этом прописывается конкретный заголовок окна, окно связывается с приложением, устанавливаются атрибуты стиля окна (`WS_OVERLAPPEDWINDOW|WS_VISIBLE`), задаются координаты верхнего левого угла и размеры окна (все — по умолчанию).

Но создание окна происходит только в памяти. Чтобы его увидеть на экране, нужно вызвать функцию отображения `ShowWindow`. *Рабочая область* окна закрашивается цветом, который устанавливается при регистрации окна. Это выполняется во время работы функции `UpdateWindow`. Обе функции получают в качестве параметра "хендл" созданного окна.

После этого окно окончательно нарисовано на экране, и главная функция выполняет цикл обработки сообщений, в котором сообщение принимается, транслируется и отправляется оконной процедуре. Текст простейшей оконной процедуры приведен в листинге 13.5.

Листинг 13.5. Оконная процедура (функция окна)

```
LRESULT CALLBACK WndProc(
HWND hwnd,    //  "хендл" окна
UINT msg,                              //  идентификатор сообщения
WPARAM wParam,                         //  дополнительный параметр
LPARAM lParam)                         //  дополнительный параметр
{  switch(msg)
   {  case WM_CREATE:            //  создается окно
      MessageBox(hwnd, "Сообщение WM_CREATE",    //  заголовок
      "Начало!",                               //  сообщение
       MB_OK|MB_ICONWARNING);                  //  значок <!>
       return 0;
       case WM_LBUTTONDOWN       //  нажата левая кнопка мыши
       MessageBox(hwnd, " Сообщение WM_LBUTTONDOWN ", "Выполняем!",
       MB_OK|MB_ICONINFORMATION);              //  значок <i>
       return 0;
       case WM_DESTROY:          //  уничтожается окно
       MessageBox(hwnd, " Сообщение WM_DESTROY ", "Конец!",
       MB_OK|MB_ICONQUESTION);                 //  значок <?>
       PostQuitMessage(0);       //  послали WM_QUIT
```

```
        return 0;
    }
      //  другие сообщения не обрабатываем
    return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Как видим, функция фактически состоит из одного оператора-переключателя. Переключающим выражением является идентификатор сообщения — целая константа. Идентификаторы сообщений имеют префикс WM. Идентификатор сообщений — это поле в структуре типа `MSG`, которая передается в качестве параметра в функциях цикла обработки сообщений. В этой структуре первые четыре поля совпадают с параметрами функции окна.

Обратите внимание, что в нашей главной программе (см. листинг 13.3) отсутствует явный вызов функции окна — эту функцию вызывает Windows при возникновении всяких событий, например, при щелчке мыши на кнопке закрытия программы. Программист должен в функции окна только определить реакцию на различные события, а управление вызовами берет на себя Windows — это одна из важнейших особенностей, на которую нужно обратить особое внимание.

Функция `MessageBox()` — очень удобное средство вывода на экран всякого рода сообщений. В данном случае мы выводим разные сообщения при создании окна, при нажатии левой кнопки мыши и при уничтожении окна. Последний параметр функции позволяет нам управлять наличием кнопок в окне сообщения и видом значка, который рисуется слева от сообщения. Например, при завершении приложения мы задали вывод знака вопроса. Вид окна сообщения представлен на рис. 13.2.

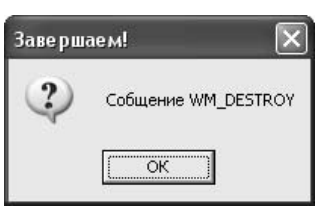

Рис. 13.2. Окно сообщения при завершении программы

При запуске программы сначала выводится окно сообщения, потом, при нажатии кнопки **OK**, выводится главное окно программы, вид которого представлен на рис. 13.3.

Если мы нажмем на левую кнопку мыши, то опять появится окно сообщения. При завершении работы (которое выполняется одним из стандартных способов Windows), после исчезновения главного окна, выводится сообщение о завершении.

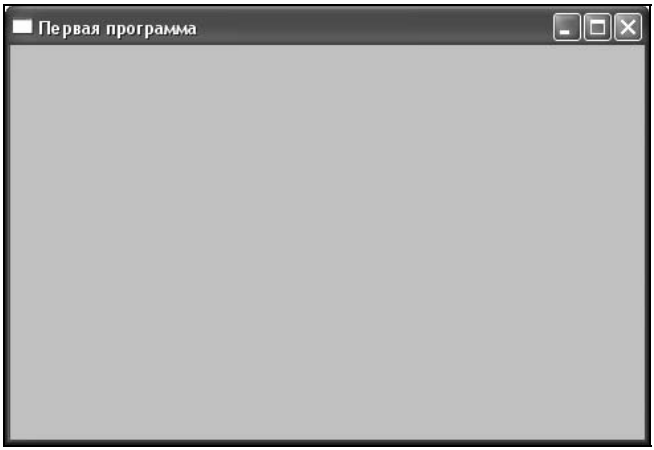

Рис. 13.3. Главное окно "пустой" программы

Интерфейс программы умножения

Пользователь, работая с программами Windows, имеет дело с многочисленными элементами управления: разнообразными кнопками, флажками, переключателями, списками, панелями управляющих элементов, разнообразными меню. Прежде чем написать программу, мы должны спроектировать ее интерфейс, т. е. те элементы управления, которые увидит и которыми будет пользоваться пользователь. В общем-то, мы делали это и раньше, но не акцентировали на этом внимание.

Во всех консольных программах, реализующих таблицу умножения (посмотрите хотя бы листинг 13.1), постоянный множитель всегда задавался пользователем с клавиатуры, а второй — вычислялся в цикле. Поэтому таблица умножения получалась всегда из 10 строк. Windows позволяет нам так спроектировать интерфейс, что вводить с клавиатуры пользователю вообще ничего не придется. Кроме того, мы можем реализовать такую версию программы, которая будет считать не всю таблицу, а только выбранные пользователем произведения. Наиболее подходящими элементами для реализации нужного интерфейса программы умножения являются:

- *флажки* — каждому множителю от 1 до 10 поставим в соответствие флажок. Если флажок установлен, значит данный множитель участвует в таблице умножения. Один специальный флажок позволяет включать в операцию все множители сразу, при установке этого флажка остальные флажки также отмечаются и блокируются. При повторной установке флажка **Все** остальные флажки разблокируются;
- *список* — второй множитель (постоянный — тот, который в прежних программах вводили с клавиатуры) будем выбирать из списка. Множители в списке будут представлены числительными;

- *кнопка* — при нажатии на кнопку выполняется умножение и вывод результатов в отдельном окне;
- *меню* — предназначено для управления флажками и для выхода из программы.

Внешний вид окна программы представлен на рис. 13.4.

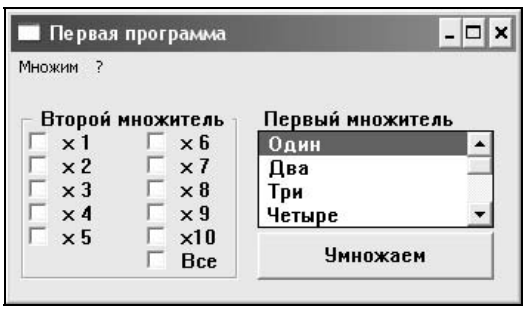

Рис. 13.4. Главное окно программы умножения

Реализация подобного интерфейса в профессиональных программах невозможна без использования ресурсов, однако программа умножения небольшая, поэтому мы поступим по-другому: реализуем весь интерфейс средствами WinAPI.

Создание меню

Меню — один из важнейших управляющих элементов в любом приложении Windows, поэтому разобраться, как создавать в программе меню — одна из первейших задач программиста.

В приложении Windows могут быть реализованы различные виды меню. В нашей "пустой" программе в левом верхнем углу окна расположено *системное меню*, предназначенное для работы с самим окном. *Пункты меню* позволяют распахивать и сворачивать окно, передвигать и изменять его размер, закрывать окно (а вместе с этим завершать программу). В любой стандартной программе Windows работает *контекстное меню*, возникающее на экране по щелчку правой кнопки мыши.

Но самым важным является *главное меню программы*, которое располагается горизонтально под заголовком окна. К каждому пункту главного меню "привязано" *выпадающее* меню из нескольких пунктов. Пусть главное меню нашей программы умножения имеет два пункта: **Множим** и **?**. Второй пункт — просто информационный и содержит единственный пункт выпадающего меню **О программе**. При выборе этого пункта выводится отдельное окно с информацией о программе. Первый пункт главного меню будет содержать выпадающее меню из трех пунктов: **Умножать по-одному**, **Умножать**

все и **Выход**. **Умножать все** эквивалентно действию флажка **Все**: отмечаются и блокируются от изменения все множители. Действие пункта **Умножать по-одному** эквивалентно повторному выбору флажка **Все** и разблокирует все флажки.

Каждому исполняемому пункту меню должно быть сопоставлено уникальное целое число, которое обычно называют идентификатором пункта меню. Это же целое число может быть использовано в качестве команды меню. По сложившейся традиции такие числа объявляют с помощью директивы `#define`:

```
#define IDM_MULT_1 1001
#define IDM_MULT_2 1002
#define IDM_EXIT 1003
#define IDM_ABOUT 1004
```

В программе четыре исполняемых пункта меню, поэтому определено четыре идентификатора.

Далее в функции окна нужно определить три "хендла": "хендл" главного меню, "хендлы" первого и второго выпадающего меню:

```
static HMENU hMenu, hFirstMenu, hAbout;
```

Нужно подчеркнуть, что важные переменные (например, "хендлы") надо объявлять в функции окна статическими, чтобы при вызовах они сохраняли постоянное значение.

Создание всех элементов управления необходимо выполнять в оконной процедуре при обработке сообщения `WM_CREATE`, поэтому добавляем туда (см. листинг 13.5) следующий фрагмент, приведенный в листинге 13.6.

Листинг 13.6. Создание меню программы

```
hMenu = CreateMenu();                  //  создание главного меню
hFirstMenu = CreatePopupMenu();        //  делаем выпадающее меню
AppendMenu(hFirstMenu, MF_STRING, IDM_MULT_1, "Умножаем по-одному");
AppendMenu(hFirstMenu, MF_STRING, IDM_MULT_2, "Умножаем все");
AppendMenu(hFirstMenu, MF_SEPARATOR, 0, NULL);     //  черта-разделитель
AppendMenu(hFirstMenu, MF_STRING, IDM_EXIT, "Выход");
  //  добавляем пункт главного меню
AppendMenu(hMenu, MF_POPUP, (UINT)hFirstMenu, "Множим");
hAbout = CreatePopupMenu();            //  создаем второе выпадающее меню
AppendMenu(hAbout, MF_STRING, IDM_ABOUT, "О программе");
  // добавляем пункт главного меню
AppendMenu(hMenu, MF_POPUP, (UINT)hAbout, "?");
SetMenu(hwnd, hMenu);                  //  связываем окно и меню
DrawMenuBar(hwnd);                     //  рисуем меню в окне
```

Как видим, сложного ничего нет, все управляется "хендлами" и константами, определенными в Windows.

Создание элементов управления

Создание флажков, списка и кнопки тоже необходимо выполнить при обработке сообщения `WM_CREATE` во время создания главного окна. Флажки, кнопки и списки являются дочерними окнами главного окна программы, поэтому создаются они той же функцией `CreateWindow`. А вид создаваемого элемента определяется классом окна и стилем. Любой управляющий элемент имеет собственный "хендл". Сначала разберемся с флажками. Это выглядит странно, но флажки считаются разновидностью кнопки, отличающейся только классом. В программе (вне функции окна) определим переменные и константы, определяющие набор наших флажков. Эти константы приведены в листинге 13.7.

Листинг 13.7. Определение флажков и кнопки

```
struct OneButton {
long style;    //  стиль флажка
char *text;    //  идентифицирующая надпись у флажка
};
const int NN = 11;
OneButton flags[NN] =    //  определение флажков
{  BS_CHECKBOX, "x 1", BS_CHECKBOX, "x 2",
   BS_CHECKBOX, "x 3", BS_CHECKBOX, "x 4",
   BS_CHECKBOX, "x 5", BS_CHECKBOX, "x 6",
   BS_CHECKBOX, "x 7", BS_CHECKBOX, "x 8",
   BS_CHECKBOX, "x 9", BS_CHECKBOX, "x10",
   BS_CHECKBOX, "Все"
};
OneButton GroupFlags = { BS_GROUPBOX, "Второй множитель" };
OneButton button = { BS_PUSHBUTTON, "Умножаем" };
```

Флажки объединим в группу, для чего и определен управляющий элемент `GroupFlags`. Определим для флажка **Все** идентификатор-константу:

```
#define ID_ALL 4010
```

Для остальных флажков определять константы не будем, но отметим, что флажкам будут соответствовать константы 4000—4009. Кроме того, в оконной процедуре определим переменные-"хендлы":

```
static HWND hGroupFlags;
static HWND hflags[NN];
```

Поскольку управляющие элементы сопровождаются текстом, то для правильного размещения надписей в окне нужно определить метрики шрифта: высоту и ширину символа. Будем использовать системный шрифт. Это делается один раз перед созданием всех управляющих элементов:

```
HDC hdc;                          //  переменная-контекст окна
TEXTMETRIC tm;                    //  переменная метрики шрифта
hdc = GetDC(hwnd);                //  получить "хендл" контекста
  //  выбрать системный шрифт
SelectObject(hdc, GetStockObject(SYSTEM_FONT));
GetTextMetrics(hdc, &tm);         //  получить метрики шрифта
cxChar = tm.tmAveCharWidth;       //  ширина символа
cyChar = tm.tmHeight;             //  высота символа
cx = tm.tmAveCharWidth;
  //  высота с межстрочным интервалом
cy = tm.tmHeight + tm.tmExternalLeading;
  //  установка цвета фона
SetBkColor(hdc, GetSysColor(COLOR_BTNFACE));
ReleaseDC(hwnd, hdc);             //  освободить контекст
```

После этого надо последовательно создать окна группы, надписи группы, флажков и кнопки (листинг 13.8). Метрики шрифта используются для того, чтобы задавать координаты и размеры окон управляющих элементов.

Листинг 13.8. Создание флажков и кнопки

```
  //  создаем группу
hGroupFlags = CreateWindow(
"button", GroupFlags.text,
WS_CHILD|WS_VISIBLE|GroupFlags.style,
cxChar, cyChar,
21 * cxChar, 7 * cyChar + 5,
hwnd, 0, hInst, NULL);
  //  создаем левый ряд флажков
cxChar = tm.tmAveCharWidth + 5;
cyChar = tm.tmHeight * 2;
for (i = 0; i < NN/2; i++)
hflags[i] = CreateWindow(
"button", flags[i].text,
WS_CHILD|WS_VISIBLE|flags[i].style,
cxChar, cyChar + tm.tmHeight*i,
60, cyChar/2,
hwnd,(HMENU)(4000 + i), hInst, NULL);
```

```
  //  создаем правый ряд флажков
cxChar = tm.tmAveCharWidth + 85;
cyChar = tm.tmHeight * 2;
for (i = 5; i < NN; i++)
hflags[i] = CreateWindow(
"button", flags[i].text,
WS_CHILD|WS_VISIBLE|flags[i].style,
cxChar, cyChar + tm.tmHeight*(i - 5),
60, cyChar/2,
hwnd,(HMENU)(4000 + i), hInst, NULL);
  //  создаем надпись группы
static HWND hLabel;
cxChar = tm.tmAveCharWidth + 160;
cyChar = tm.tmHeight;
hLabel = CreateWindow(
"static", " Второй множитель ",
WS_CHILD|WS_VISIBLE,
cxChar, cyChar,
160, cyChar + tm.tmHeight/2,
hwnd,0,hInst,NULL);
  //  создаем кнопку
static HWND hButton;
cxChar = tm.tmAveCharWidth + 160;
cyChar = tm.tmHeight*6 + 5;
hButton = CreateWindow(
"button", button.text,
WS_CHILD|WS_VISIBLE|button.style,
cxChar, cyChar,
160, 2 * tm.tmHeight,
hwnd,(HMENU)ID_BUTTON, hInst, NULL);
```

Как видите, все создается единообразно. Каждый элемент имеет свой "хендл" и связывается с главным окном и приложением по соответствующим "хендлам" главного окна и приложения. Надпись — это статический управляющий элемент (класс окна `static`). Для кнопки определена константа-идентификатор `ID_BUTTON`:

```
#define ID_BUTTON 2000
```

Список создается совершенно аналогично: определяются константы-числительные, константа-идентификатор и переменная-"хендл":

```
const char *szList[10] = { "Один", "Два", "Три",
                           "Четыре", "Пять", "Шесть",
                           "Семь", "Восемь", "Девять", "Десять"
                         };
```

```
#define ID_LISTBOX 3000
static HWND hListBox;
```

Список тоже создается как окно и затем наполняется строками-числительными:

```
cxChar = tm.tmAveCharWidth + 160;
cyChar = 2 * tm.tmHeight;
hListBox = CreateWindow(
"listbox", NULL,
WS_CHILD|WS_VISIBLE|WS_VSCROLL|LBS_NOTIFY|WS_BORDER,
cxChar, cyChar,
160, 5 * tm.tmHeight,
hwnd,(HMENU)ID_LISTBOX,hInst,NULL);
for (i = 0; i < 10; ++i)    //  наполнение списка
SendMessage(hListBox, LB_ADDSTRING, 0, (LPARAM) szList[i]);
```

Тут же надо отметить первый элемент в списке:

```
SendMessage(hListBox, LB_SETCURSEL,0,0);
```

В завершение обработки сообщения `WM_CREATE` нужно прописать оператор возврата:

```
return 0;
```

Обработка сообщений

Далее в функции окна (см. листинг 13.5) надо определить обработку сообщения `WM_COMMAND`. Это основное сообщение, которое практически и определяет функциональность приложения. "Внутри" мы должны определить оператор-переключатель, в котором в качестве альтернатив заданы уже определенные нами константы-идентификаторы. Переключающее выражение — это младшая часть параметра `wParam`. Каждая альтернатива должна заканчиваться оператором возврата. Текст фрагмента для обработки команд приведен в листинге 13.9.

Листинг 13.9. Обработка команд

```
case WM_COMMAND:
switch(LOWORD(wParam))
{  case IDM_EXIT:      //  выход
   SendMessage(hwnd, WM_CLOSE, 0, 0L);
   return 0;
   case IDM_MULT_1:    //  умножаем по-одному
   SendMessage(hflags[10], BM_SETCHECK, FALSE, 0L);
```

```
for (i = 0; i < 10; ++i)
{  SendMessage(hflags[i], BM_SETCHECK, FALSE, 0L);
   EnableWindow(hflags[i], TRUE);
   m2[i] = result[i] = 0;
}
return 0;
case ID_ALL:        //  флажок "Все"
case IDM_MULT_2:    //  умножаем все
  //  включен ли флажок ?
if (!SendMessage(hflags[10], BM_GETCHECK, 0, 0L))
{  SendMessage(hflags[10], BM_SETCHECK, TRUE, 0L);
   for (i = 0; i < 10; ++i)           //  отмечаем флажки
   {  SendMessage(hflags[i], BM_SETCHECK, TRUE, 0L);
      EnableWindow(hflags[i], FALSE); //  блокируем флажки
      m2[i] = i + 1;                  //  заполняем множители
   }
}
else
{  SendMessage(hflags[10], BM_SETCHECK, FALSE, 0L);
   for (i = 0; i < 10; ++i)           //  снимаем флажки
   {  SendMessage(hflags[i], BM_SETCHECK, FALSE, 0L);
   EnableWindow(hflags[i], TRUE);     //  открываем флажки
   m2[i] = result[i] = 0;             //  очищаем множители
   }
}
return 0;
case IDM_ABOUT:                       //  вывести окно "О программе"
if(!IsWindow(hAbout))                 //  если окна еще нет на экране
hAbout = CreateWindow(
szClassAbout, "О программе",
WS_POPUPWINDOW|WS_VISIBLE|WS_CAPTION,
CW_USEDEFAULT,CW_USEDEFAULT,
strlen("Таблица множения") * cx * 1.5, 100,
hwnd, 0, hInst, NULL);
return 0;
case ID_BUTTON:             //  выполнить умножение
for (i = 0; i < 10; ++i)
{ result[i] = m1 * m2[i]; if(result[i]) All = true; }
if (All)                    //  если отмечен хоть один флажок
{  if(!IsWindow(hMulti))    //  если окна нет на экране
   hMulti = CreateWindow(szClassMulti,
   "Таблица умножения",
   WS_POPUPWINDOW|WS_VISIBLE|WS_CAPTION,
```

```
      CW_USEDEFAULT,CW_USEDEFAULT,
      strlen(" Таблица умножения") * cx * 1.5, cy * 12,
      hwnd, 0, hInst, NULL);
   }
   else
   MessageBox(
   hwnd, " Не выбрано ни одного множителя!",
   "Ошибка",
   MB_OK|MB_ICONERROR);
   return 0;
   case 4000:case 4001:case 4002:    //  флажки
   case 4003:case 4004:case 4005:
   case 4006:case 4007:case 4008:case 4009:
   i = LOWORD(wParam) - 4000; m2[i] = i + 1;
   if(!SendMessage(hflags[i], BM_GETCHECK, 0, 0L))
   SendMessage(hflags[i], BM_SETCHECK, TRUE, 0L);
   else
   SendMessage(hflags[i], BM_SETCHECK, FALSE, 0L);
   return 0;
   case ID_LISTBOX:                  //  список
   if(HIWORD(wParam) == LBN_SELCHANGE)
   { m1 = SendMessage(hListBox, LB_GETCURSEL, 0, 0) + 1; }
   return 0;
}
```

Первое, на что надо обратить внимание, — выполнение действий с помощью посылки сообщений, причем элемент, которому сообщение посылается, определяется посредством соответствующего "хендла". Посылка сообщений выполняется функцией `SendMessage`, в которой указывается "хендл", нужная константа-сообщение и другие параметры, сопутствующие сообщению. Например, опрос флажка **Все** выполняется таким вызовом:

```
SendMessage(hflags[10], BM_GETCHECK, 0, 0L)
```

Установка и блокировка флажков осуществляются так:

```
SendMessage(hflags[i], BM_SETCHECK, TRUE, 0L);
EnableWindow(hflags[i], FALSE);
```

При выполнении команды `ID_ABOUT` требуется открыть новое окно и вывести в нем небольшой текст. При обработке сообщения это окно создается как подчиненное выпадающее (`popupwindow`) окно. Причем сначала проверяется, существует ли оно. Окно такого же вида создается и для вывода таблицы умножения. Оба окна должны иметь собственные "хендлы", и в главной функции должны быть зарегистрированы соответствующие классы окон:

```
if (!RegClass(hI, WndAbout, szClassAbout)) return FALSE;
if (!RegClass(hI, WndMulti, szClassMulti)) return FALSE;
```

Каждое из окон должно иметь собственную функцию окна. В этой функции должно обрабатываться сообщение `WM_PAINT`. Обработка этого сообщения тоже должна подчиняться жесткой дисциплине. Дело в том, что текст в окне рисуется, поэтому для этого требуется использовать определенную часть WinAPI, которая по-английски называется *Graphics Device Interface* (*GDI*) — интерфейс графического устройства. В листинге 13.10 приведена функция окна **О программе**.

Листинг 13.10. Функция окна О программе

```
LRESULT CALLBACK WndAbout(
HWND hwnd,
UINT msg,
WPARAM wParam,
LPARAM lParam)
{  HDC hdc;           //  переменная — контекст
   PAINTSTRUCT pc;    //  информация о рисовании
   RECT rect;         //  прямоугольник рабочей области
   switch(msg)
   {  case WM_PAINT:
      hdc = BeginPaint(hwnd, &pc);
      GetClientRect(hwnd, &rect);
      SetBkColor(hdc, GetSysColor(COLOR_BTNFACE));
      DrawText(hdc, "\n Таблица умножения\n учебная программа ",
      -1, &rect, DT_CENTER|DT_VCENTER);
      EndPaint(hwnd, &pc);
      return 0;
   }
   return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Сначала функция `BeginPaint` выдает *контекст* устройства, затем с помощью функции `GetClientRect` получаем рабочую область окна, устанавливаем фон рабочей области такой же, как системный цвет управляющих элементов, с помощью функции `SetBkColor`.

Рисование текста выполняется функцией `DrawText`, параметрами которой являются строка и константы, задающие выравнивание. Строка делится на отдельные подстроки символом `'\n'`. Каждая из подстрок подчиняется заданному выравниванию.

Завершение обработки должно завершаться функцией `EndPaint()`. Вид окна **О программе** представлен на рис. 13.5.

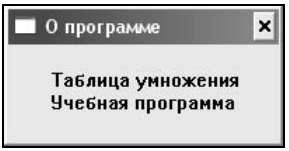

Рис. 13.5. Вид окна **О программе**

При обработке сообщения, поступившего при нажатии кнопки, сначала вычисляются произведения. Если есть хоть один ненулевой результат, то создается окно с заголовком **Таблица умножения**. В функции окна, текст которой приведен в листинге 13.11, тоже обрабатывается сообщение `WM_PAINT`.

Листинг 13.11. Функция окна Таблица умножения

```
LRESULT CALLBACK WndMulti(
HWND hwnd,
UINT msg,
WPARAM wParam,
LPARAM lParam)
{  HDC hdc;                          //  переменная-контекст устройства
   PAINTSTRUCT pc;                   //  информация для рисования
   RECT rect;
   char szString[200] = {0};         //  объединенная строка
   char szBuffer[20] = {0};          //  отдельное произведение
   switch(msg)
  {  case WM_PAINT:
     for (int i = 0; i < 10; ++i)    //  подготовка строки для вывода
     {  if (m2[i])
        {  sprintf(szBuffer, "%2d x %2d = %2d\n", m2[i], m1,result[i]);
           strcat(szString, szBuffer);
        }
     }
   hdc = BeginPaint(hwnd, &pc);
   GetClientRect(hwnd, &rect);
   SetBkColor(hdc, GetSysColor(COLOR_BTNFACE));
   DrawText(hdc, szString, -1, &rect, DT_CENTER|DT_VCENTER);
   EndPaint(hwnd, &pc);
   return 0;
   }
   return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Сначала готовится строка вывода. Для перевода одного произведения в строковый вид мы использовали один из простых давно известных спосо-

бов, реализованных в стандартной библиотеке ввода/вывода C — stdio.h. Затем отдельное произведение "прицепляется" к выводимой строке. После создания полной строки выполняется рисование текста абсолютно точно так же, как и в функции окна **О программе**.

Таким образом, в окно **Таблица умножения** может выводиться как полная таблица (рис. 13.6), так и частичная (рис. 13.7).

Рис. 13.6. Полная таблица умножения

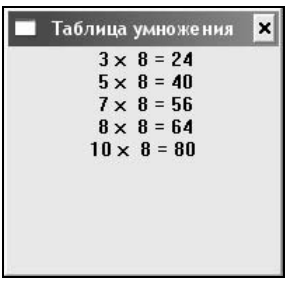

Рис. 13.7. Частичная таблица умножения

Стоила ли игра свеч

Собрав все описанные в *предыдущих разделах* примеры, получим довольно большую программу, которая реально выполняет очень простую работу. При этом мы затронули очень небольшую часть возможностей, предоставляемых Windows. Стоило ли для этого использовать WinAPI? Как мы только что убедились, программирование с применением WinAPI — достаточно сложный процесс. Кроме четкой структуры программы, часто требуется придерживаться жесткой дисциплины использования многих функций. Необходимо знание не только принципов программирования для Windows, но и огромного количества мелких деталей, без которых программа не работает. Сотни стандартных идентификаторов, десятки префиксов — все это требует повышенного внимания. Например, в процессе отладки рассмотренного приложения я довольно часто экспериментировал с организацией дочерних окон и их цветами. И в один момент я забыл поставить скобки (выделены полужирным) в операторе установки цвета фона в функции регистрации класса окна:

```
wc.hbrBackground = (HBRUSH)(COLOR_BTNFACE + 1);
```

В результате фон окна оказался совсем не таким, на который я рассчитывал. Я "убил" на поиск ошибки около часа, пока не сообразил, в чем дело. Таких примеров можно привести много. Недаром Чарльз Петцольд в своей книге "Программирование для Windows 95" назвал первую программу "HelloWin", тонко намекая на праздник "нечистой силы".

И все-таки, несмотря на многочисленные сложности, мы получаем несомненные выгоды. Во-первых, мы фактически создали каркас своих будущих программ. Причем, этот каркас уже очень много "умеет" и "знает": окна передвигаются по экрану, изменяют размеры, распахиваются и закрываются. У каждого окна есть заголовок и системное меню — и нам не пришлось ничего этого программировать. Да и внешний вид элементов управления уже весь заложен "внутрь".

Во-вторых, всегда полезно знать, что скрывается "под" какой-нибудь объектно-ориентированной библиотекой, например MFC или VCL — это помогает писать более качественные программы.

На этом закончим знакомство с огромным "айсбергом", которым является WinAPI.

ГЛАВА 14

Быстрая разработка приложений

Как мы могли убедиться в *гл. 13*, разработка приложений для Windows с использованием WinAPI требует знания огромного количества технических деталей. В этом море "утонул" не один начинающий программист. Осознав существующую проблему, фирма Borland предложила новый тип интегрированной среды, предназначенной для облегчения создания оконного интерфейса. Сначала это была среда для Pascal под названием Delphi, а потом и для C++ — C++ Biulder. В составе интегрированной среды реализована мощная объектно-ориентированная библиотека VCL (Visual Component Library), предназначенная, в первую очередь, для программирования интерфейса приложения. Давайте попробуем реализовать программу вычисления таблицы умножения в этой интегрированной среде. Последовательность действий представим как последовательность шагов, выполняемых программистом.

Шаг 1

Запускаем C++ Builder. Он обычно (если никто ничего не менял в настройках) предлагает нам созданный им по умолчанию проект с одним пустым окном. Это окно называется *формой*.

Проект можно уже компилировать (<Ctrl>+<F9>) и запустить (<F9>) — в результате получим приложение, "умеющее" делать все, что должно "уметь" любое окно Windows: сворачиваться и разворачиваться, перемещаться и изменять размеры.

Сравнивая с WinAPI можно сказать, что лишнего ничего писать не требуется. Создание окна, регистрация его класса, сам цикл обработки сообщений спрятан в глубине VCL и программист его не увидит, да и смысла в этом особого нет.

Шаг 2

Попробуем спроектировать (наполнить содержимым) интерфейс главной формы нашего приложения. Делается это с помощью панели компонентов (рис. 14.1).

Рис. 14.1. Панель компонентов

Сначала поместим на форму элементы управления `GroupBox` (две группы — **Первый множитель**, **Второй множитель**) и `Button` (кнопку) из вкладки **Standard**.

Далее в группу **Первый множитель** добавим 11 элементов типа `CheckBox` (флажков), а в группу **Второй множитель** — элемент типа `ComboBox` (выпадающий список). Элементы управления можно свободно перемещать и менять их размеры с помощью мыши и клавиатуры. Можно объединить элементы в группу, выделяя их левой кнопкой мыши при нажатой клавише <Shift>. Тогда операции перемещения и изменения размера можно выполнять с группой.

В итоге внешний вид формы должен получиться таким, как представлено на рис. 14.2.

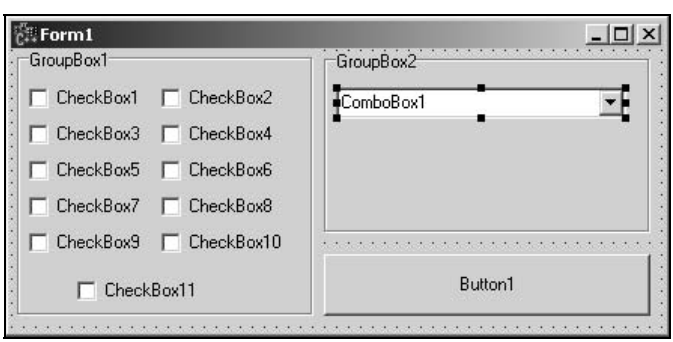

Рис. 14.2. Форма с элементами управления

Теперь поработаем со свойствами. Для работы со свойствами в режиме разработки (Design Mode) служит окно **Object Inspector**, внешний вид которого представлен на рис. 14.3.

В нем отображаются и изменяются свойства выделенного *компонента* или общие свойства группы выделенных компонентов.

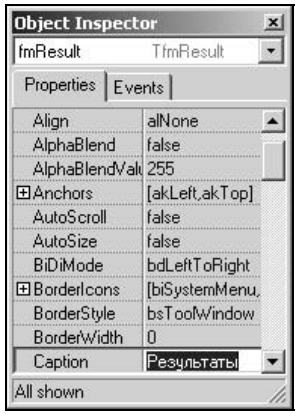

Рис. 14.3. Окно **Object Inspector**

Компоненты могут быть *визуальными* — это все видимые на экране элементы во время работы программы. К ним, например, относятся элементы управления, сама форма. С помощью этих компонентов реализуют интерфейсную часть проекта. В системе есть и *не визуальные компоненты*, например, таймер, компоненты доступа к базам данных. Эти компоненты реализуют функциональность проекта, но не отображаются. Пока мы имеем дело только с визуальными компонентами.

Свойства визуальных компонентов чаще всего определяют их внешний вид. Например, свойство `Caption` (англ. — заглавие) для формы определяет текст в заголовке окна. То же свойство для флажков и кнопок — текстовую надпись. Некоторые наиболее часто используемые свойства означают следующее:

- `Font` — определяет шрифт, которым выводится текст;
- `Align` — выравнивает компонент в форме;
- `TabOrder` — определяет порядок перехода между элементами управления по клавише <Tab>;
- `TabStop` — разрешает или запрещает перемещение по клавише <Tab>;
- `Hint` — определяет строку, служащую контекстной подсказкой;
- `Enabled` — разрешает/запрещает использование элемента управления;
- `Visible` — скрывает/показывает элемент управления;
- группа свойств `Left`, `Top`, `Width`, `Height` — определяют позицию и размеры элемента управления в пикселах.

Особым свойством, присутствующим у любого компонента, является *имя* (`Name`). Имя определяет наименование переменной, инкапсулирующей данный компонент. Например, если у нас на какой-то форме есть флажок с именем `checkbox1` — это означает, что внутри класса формы есть объявление поля `TCheckBox* checkbox1`.

Существует еще одно интересное свойство `Tag` (целое число). Оно есть у большинства компонентов, и предназначено для использования программистом, например для идентификации групп элементов управления.

Есть также ряд свойств, необходимых для функционирования каждого конкретного вида элемента управления. Например, у флажков есть свойство `Checked`, принимающее значение `true`, если галочка установлена, и `false`, если нет.

У элемента `ComboBox` есть свойство `Items`. Чтобы отредактировать его, нужно нажать на кнопку справа. В результате появится редактор списка строк. Каждая введенная строка станет элементом нашего выпадающего списка. Кроме того, есть свойства `Text` и `ItemIndex`, определяющие соответственно выбранную строку и ее номер (позицию) в списке. Чтобы подробно узнать назначение того или иного свойства, пользуйтесь справкой.

Настроим свойства элементов управления, помещенных в форму. После настройки форма приобретает вид, представленный на рис. 14.4.

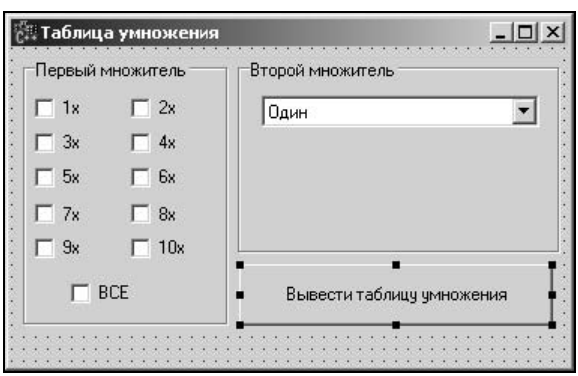

Рис. 14.4. Форма с настроенными компонентами

Теперь можно добавить меню. После добавления компонента `TMainMenu` нужно щелкнуть по нему два раза — появится редактор меню, внешний вид которого можно видеть на рис. 14.5.

С его помощью можно легко и просто организовать меню приложения. Меню нашей программы будет включать два пункта главного меню: основной пункт **Умножение** и пункт **?**, который просто выводит окно **О программе**. Пункт меню **Умножение** включает следующие пункты выпадающего меню:

- **Частичная** — вычисление таблицы умножения для одиночного множителя;
- **Полная** — вычисление полной таблицы умножения;
- **Выход** — завершение работы программы.

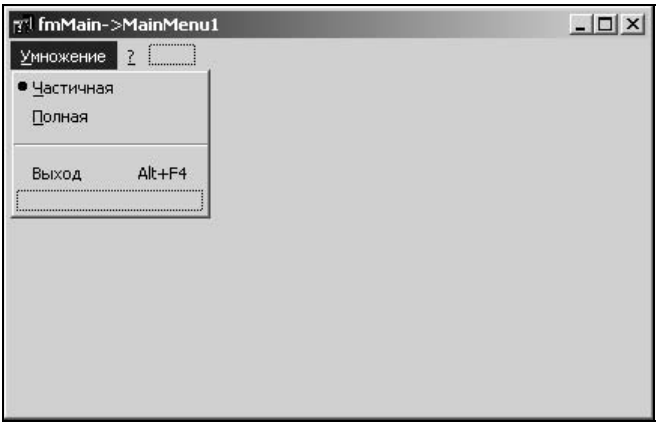

Рис. 14.5. Редактор меню

Учитывайте, что знак `&` перед символом в свойстве `Caption` делает символ "горячим". В меню символ станет подчеркнутым, и можно выбрать этот пункт меню, нажав комбинацию клавиш <Alt>+<Символ>. Для пунктов также можно назначать функциональные "горячие клавиши" (свойство `Shortcut`). Свойство `RadioItem` позволяет сделать пункт меню отмечаемым (флажком).

Шаг 3

Окно (см. рис. 14.5) уже обладает базовой функциональностью: флажки устанавливаются и сбрасываются, элементы выбираются из списка и т. д. Теперь реализуем функциональность флажка **ВСЕ**: требуется, чтобы при установке флажка **ВСЕ** остальные флажки тоже устанавливались и блокировались для редактирования, а при снятии флажка — разблокировались. Кроме того, пусть на установку/сброс флажка **ВСЕ** влияет выбор пунктов меню **Частичная**/**Полная**. При выборе пункта **Полная** выполняются те же действия, что и при установке флажка **ВСЕ**: остальные флажки тоже устанавливаются и блокируются. В пункте **Частичная** реализуются те же действия, что и при снятии флажка **ВСЕ**.

Для этого нужно определить *события* — их можно увидеть на вкладке **Events** окна **Object Inspector**, представленного на рис. 14.6.

Необходимо определить обработчик события флажка **ВСЕ** `OnClick`, который будет срабатывать по щелчку левой кнопки мыши. Это можно сделать, дважды щелкнув на соответствующей строке окна **Object Inspector** или на флажке в форме. В результате появится окно редактора кода с заготовкой функции-обработчика события (рис. 14.7).

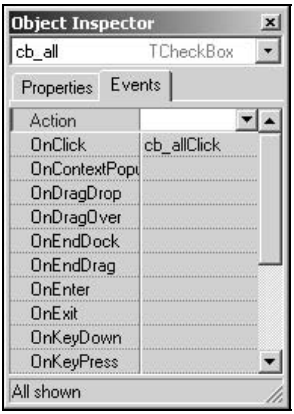

Рис. 14.6. События в окне **Object Inspector**

Все обработчики событий по умолчанию — члены класса соответствующей формы, поэтому в их области видимости находятся все поля и методы этой формы.

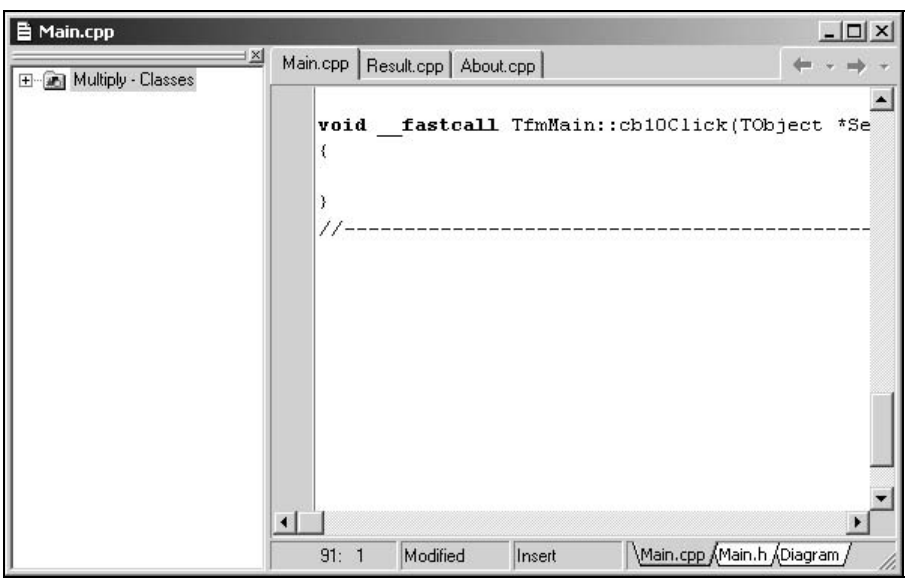

Рис. 14.7. Окно редактора кода с заготовкой обработчика события

Первое, что мы сделаем, — установим выбранным соответствующий пункт меню.

```
N2->Checked = !(cb_all->Checked);
N3->Checked = cb_all->Checked;
```

N2 — имя пункта меню с названием **Частичная**, N3 — с названием **Полная**. Обработчик события OnClick срабатывает как при установке, так и при сбросе флажка **ВСЕ**, поэтому мы используем для получения информации об его состоянии выражение cb_all->Checked.

Далее, в зависимости от состояния переменной cb_all, нам надо либо запретить, либо разрешить устанавливать остальные элементы типа CheckBox в форме. Это можно сделать "в лоб", как приведено в листинге 14.1.

Листинг 14.1. Запрещение/разрешение установки флажков в форме (версия 1)

```
if (cb_all->Checked) {
cb1->Checked = true;
cb1->Enabled = false;
cb2->Checked = true;
cb2->Enabled = false;
cb3->Checked = true;
cb3->Enabled = false;
cb4->Checked = true;
cb4->Enabled = false;
cb5->Checked = true;
cb5->Enabled = false;
cb6->Checked = true;
cb6->Enabled = false;
cb7->Checked = true;
cb7->Enabled = false;
cb8->Checked = true;
cb8->Enabled = false;
cb9->Checked = true;
cb9->Enabled = false;
cb10->Checked = true;
cb10->Enabled = false;
}
else {
cb1->Enabled = true;
cb2->Enabled = true;
cb3->Enabled = true;
cb4->Enabled = true;
cb5->Enabled = true;
cb6->Enabled = true;
cb7->Enabled = true;
cb8->Enabled = true;
cb9->Enabled = true;
cb10->Enabled = true;
}
```

Но можно написать и короче, как приведено в листинге 14.2.

Листинг 14.2. Запрещение/разрешение установки флажков в форме (версия 2)

```
if (cb_all->Checked) {
for (int i = 1; i < 11; i++) {
TCheckBox* cb_control =
(TCheckBox*)fmMain->FindComponent("cb" + IntToStr(i));
cb_control->Checked = true;
cb_control->Enabled = false;
}
}
else {
for (int i = 1; i < 11; i++) {
TCheckBox* cb_control =
(TCheckBox*)FindComponent("cb" + IntToStr(i));
cb_control->Enabled = true;
}
}
```

Функция `FindComponent` (член класса `TForm`) ищет на форме компонент с указанным именем. Приведение типов надо делать явно, потому что возвращается тип `TComponent*` (базовый класс для всех компонентов). Естественно, что код сработает, если флажки имеют имена вида `cbx`, где `x` изменяется от 1 до 10.

Последние два штриха — добавим обработчики событий для пунктов меню, написав в соответствующих функциях-заготовках:

```
cb_all->Checked = false;
```

и

```
cb_all->Checked = true;
```

соответственно.

Теперь при установке флажка **ВСЕ** окно нашей программы выглядит так, как представлено на рис. 14.8.

Шаг 4

Добавим окно **О программе**. Для этого выполним команды **File | New | New Form**. Появится новая форма, на нее надо "повесить" несколько элементов управления типа `Label` и кнопку. Тогда наша форма будет выглядеть так, как показано на рис. 14.9.

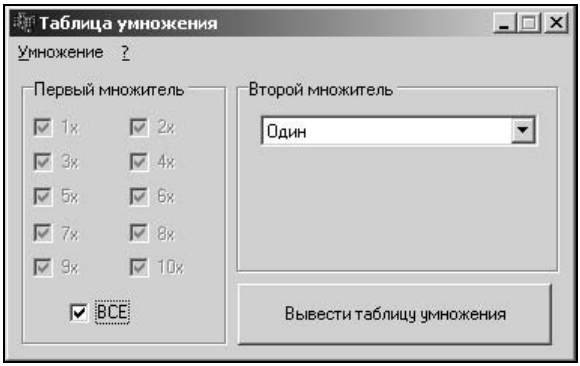

Рис. 14.8. Вид окна **Таблица умножения** при установке флажка **ВСЕ**

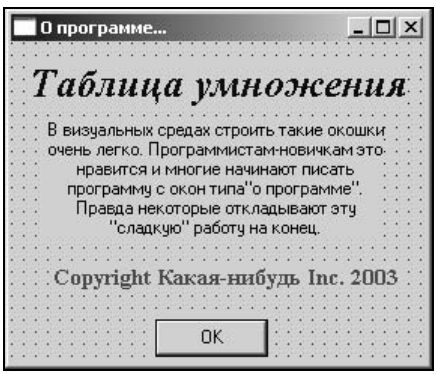

Рис. 14.9. Форма **О программе**

Заметьте, что каждая форма состоит из трех файлов с расширениями h, cpp (класс формы) и dfm (ресурс-описание). Подключим h-файл окна **О программе** к основной форме. В обработчике пункта меню **?** напишем следующую строчку:

```
fmAbout->ShowModal();
```

Этот метод показывает форму **О программе** в модальном режиме: остальные окна приложения недоступны, пока видно это окно.

В обработчике на нажатие кнопки **OK** напишем вызов метода `Close()`. Он закрывает форму.

Шаг 5

Создадим форму вывода результатов с двумя элементами управления: `Grid` (таблица, закладка) и кнопкой. Вид окна представлен на рис. 14.10.

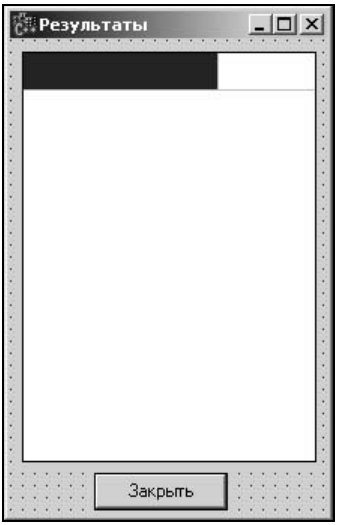

Рис. 14.10. Окно **Результаты**

Обработчик на нажатие кнопки здесь такой же, как и для формы **О программе**. Подключим h-файл окна результатов к основной форме.

До сих пор мы имели дело с автоматически создаваемыми формами. Мы не вызывали конструктор формы перед обращением к ее методам и свойствам. Форма создавалась автоматически при загрузке программы, а далее показывалась или скрывалась. Для того чтобы запретить автоматическое создание формы, надо выбрать пункт меню интегрированной среды **Projects | Options** (вид окна опций для нашего файла представлен на рис. 14.11).

Нужно убрать форму результатов из списка автоматически создаваемых форм в окне опций на вкладке **Forms**.

Теперь в обработчике кнопки **Вывести таблицу умножения** напишем код, приведенный в листинге 14.3.

Листинг 14.3. Создание и связывание формы результатов с основной формой

```
int y, k = 0;
TfmResult* resfrm = new TfmResult(this);
y = ComboBox1->ItemIndex+1;
for (int i = 1; i < 11; i++){
TCheckBox* cb_control = (TCheckBox*)FindComponent("cb" + IntToStr(i));
if (cb_control->Checked){
resfrm->AddResultLine(i, y);
k++;
}
}
```

```
if (k > 0) resfrm->Show();
else MessageBox( this->Handle, "", "", MB_OK | MB_ICONEXCLAMATION);
```

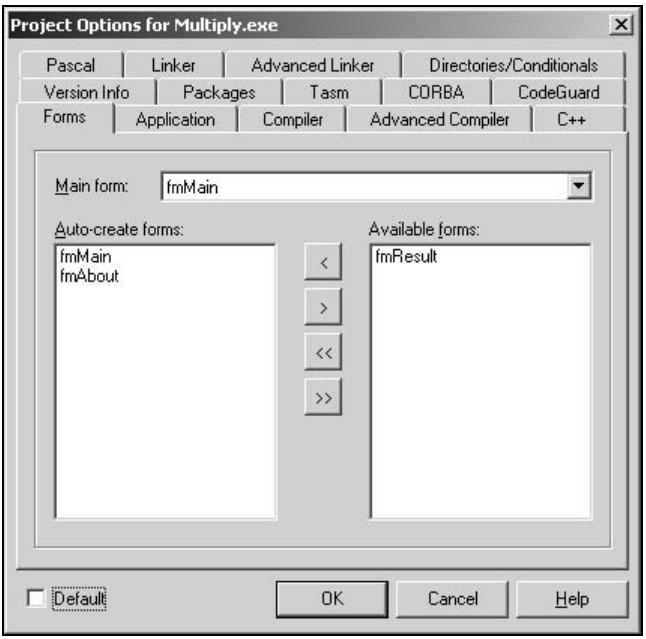

Рис. 14.11. Окно опций для установки режимов работы

Первое, что мы делаем, — создаем новую форму, как дочернюю от текущей. Далее получаем индекс выбранного элемента выпадающего списка и отождествляем его с множителем. Знакомым нам способом проверяем установку флажков, и если проверяемый флажок установлен, вызываем функцию созданной нами формы под названием `AddResultLine(i, y)` — ее нам предстоит реализовать в форме результатов.

Кроме того, мы ведем подсчет установленных флажков, если ни один флажок не был установлен, то вызываем функцию WinAPI `MessageBox`.

Заметьте, что созданная форма работает не в модальном режиме, а в обычном. Более того, тот факт, что мы каждый раз создаем новое окно, позволяет нам выводить несколько окон результатов одновременно, как показано на рис. 14.12.

Реализуем функцию `AddResultLine`. В C++ Builder есть мастер, упрощающий процесс добавления новой функции в класс. Окно мастера добавления метода представлено на рис. 14.13.

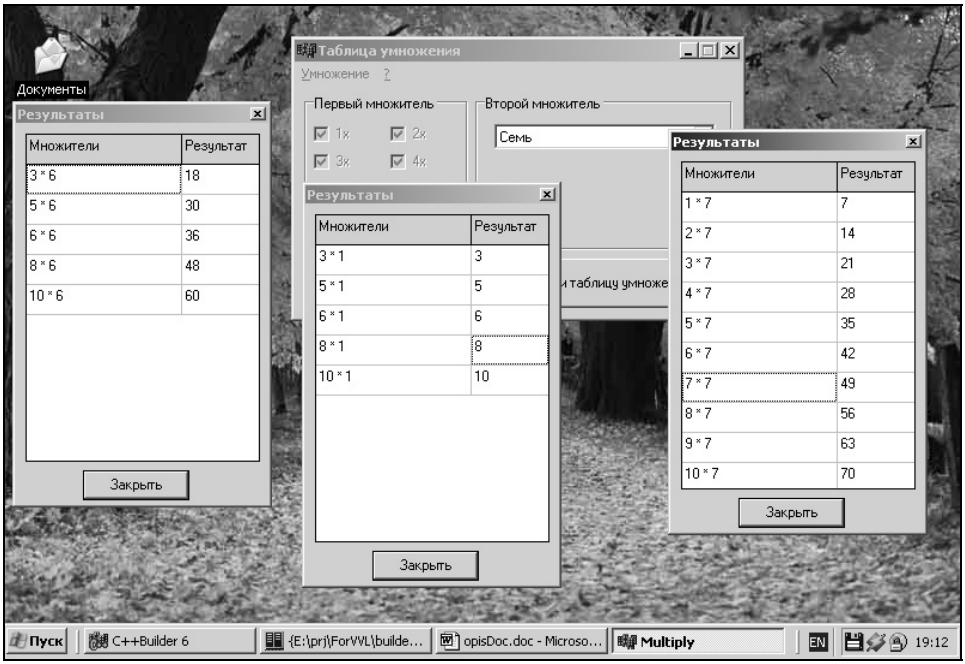

Рис. 14.12. Несколько окон результатов

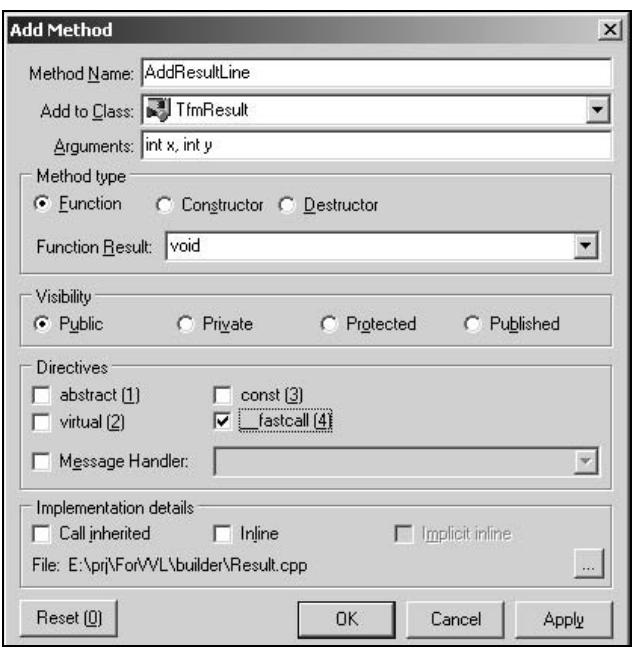

Рис. 14.13. Диалоговое окно **Add Method**

Код, который нам нужно добавить, выглядит так:

```
StringGrid->RowCount = StringGrid->RowCount+1;
StringGrid->Cells[0][StringGrid->RowCount-1] =
IntToStr(x) + " * " + IntToStr(y);
StringGrid->Cells[1][StringGrid->RowCount-1] = IntToStr(x*y);
```

С помощью этого фрагмента мы увеличиваем количество строк в таблице и записываем в добавленные ячейки выражение и результат перемножения.

Последний штрих: добавим обработчик `OnFormShow` для формы результатов. Он создает нам шапку таблицы умножения в форме результатов:

```
StringGrid->Cells[0][0] = "Выражение";
StringGrid->Cells[1][0] = "Результат";
StringGrid->FixedRows = 1;
```

На этом закончим нашу очень краткую экскурсию в технологию быстрой разработки приложений.

Заключение

Я поздравляю вас, читатель — вы добрались до последней страницы этой книги! Завершая чтение, всегда полезно задать себе вопрос, что нового удалось из нее почерпнуть. Я надеюсь, что ни время, ни деньги, потраченные читателем на эту книгу, не оказались затраченными впустую. Однако книга только приоткрывает дверь в прекрасный и огромный дворец C++ — многие обширные и глубокие темы только затронуты. По каждой из них можно написать (или уже написана) отдельную книгу в два-три раза толще той, что вы держите в руках. В этом "море" информации можно просто-напросто утонуть, если не иметь ориентиров. Поэтому полезно наметить путь, по которому вы, читатель, двинетесь дальше.

В *приложении 4* перечислено много книг, однако мне, как и всякому другому программисту, одни нравятся больше, другие меньше. Некоторые из них обязательно должны стоять на полке каждого программиста, независимо от того, на каком языке он работает.

Для изучения общих основ объектно-ориентированного программирования хороша книга Т. Бадда [4] и ставшая уже "библией" книга Гради Буча [8]. Такой же "библией" является книга "банды четырех" про паттерны проектирования [10]. Много хороших книг написано про алгоритмы и структуры данных [7, 9, 14, 39, 40, 41, 45]. Но мне наиболее импонируют книга Вирта [9], хотя она и написана с использованием языка Pascal, и книга Роберта Сэджвика [39, 40]. В ней, а также в книге [14] прекрасно изложена рекурсия.

Если вас интересует программирование в интегрированной среде Visual C++ 6, можно прочитать [19, 26, 28, 46], из которых "библией", конечно, является [19].

Несколько книг [3, 32, 36] рассказывают об устройстве компиляторов. Самая лучшая из них — первая. Программирование для Windows лучше начинать с книги Харта [43], после чего можно браться за более фундаментальные труды Рихтера [33] и Фень Юаня [50].

Ну и практически обязательно каждый квалифицированный программист должен прочитать книги Б. Страуструпа [37, 38], Мейерса [23, 24, 25], серию "C++ In-Depth" [1, 15, 20, 35], издаваемую под редакцией того же Страуструпа. Для глубокого изучения указателей просто необходима книга Элджера [49]. Стандартную библиотеку STL лучше начинать изучать по книге Аммерааля [2], но и книга Халперна [42] тоже неплоха. Скоро в издательстве "Питер" будет издан фундаментальный труд Н. Джосатиса по STL.

Ну и, конечно, читайте стандарт C++. Жизнь не стоит на месте, язык C++ развивается — уже официально утвержден новый стандарт (в списке адресов Интернета указан адрес комитета по стандартизации). Прежний стандарт за рубежом вышел в печатном виде.

Читая книги, не забывайте, однако, что в конечном счете единственный способ совершенствования в программировании — писать программы. Удачи вам!

ЧАСТЬ IV

Приложения

Приложение 1. Системы фирмы Borland

Приложение 2. Интегрированная среда Microsoft Visual C++ 6

Приложение 3. Ресурсы C++ в Интернете

Приложение 4. Список литературы

ПРИЛОЖЕНИЕ 1

Системы фирмы Borland

Современная система программирования — это довольно большой программный продукт, в котором объединяются текстовый редактор, компилятор, компоновщик, мейкер, отладчик, подсистема управления файлами и справочная система. Поэтому такие программы называются *интегрированными средами разработки* (Integrated Development Environment — *IDE*). Чтобы квалифицированно писать программы, программист, кроме самого языка программирования, обязан знать возможности интегрированной среды. В общем-то, возможности одной среды программирования мало чем отличаются от другой, однако, чтобы понять это, надо попробовать создать одну и ту же программу хотя бы в двух системах. В приложении приводятся минимально необходимые сведения по трем системам: Borland C++ 3.1, Borland C++ 5 и Borland C++ Builder 6. Начинать с этих систем удобно, т. к. большинство действий пользователя одинаково во всех этих системах. То же самое можно сказать и о "горячих клавишах".

Borland C++ 3.1

Система создана более 10 лет назад (в 1992 году), однако до сих пор довольно активно используется как в обучении, так и в реальном программировании. К тому же система весьма проста, и изучение ее послужит хорошим трамплином в освоении других IDE фирмы Borland (и других фирм). Система является многооконной и позволяет выполнять со своими окнами все, что обычно можно делать с окнами в Windows: перемещать, изменять размеры, распахивать и сворачивать, располагать "каскадом" или "плиткой" и закрывать. Система имеет собственный (независимый от Windows) буфер обмена.

При установке на жесткий диск система предлагает установить версии системы для DOS и для Windows. Следует выбрать установку только для DOS, т. к. версия для Windows сильно устарела.

После установки система будет записана в каталоге BC (можно изменить при установке). Внутри находятся несколько подкаталогов, важнейшими из которых являются:

- **Bin** — здесь находятся все программы интегрированной среды;
- **Include** — в этом каталоге размещены системные подключаемые файлы;
- **Lib** — этот каталог содержит объектные файлы (файлы с расширением obj) и библиотеки (файлы с расширением lib) стандартных функций, которые компоновщик подключает к вашей программе на этапе компоновки;
- **Examples** — содержит многочисленные примеры программ для изучения;
- **Doc** — включает ряд текстовых файлов с руководствами по некоторым программам интегрированной среды;
- **TVision** — здесь находится объектно-ориентированная библиотека оконного интерфейса — предшественник известных библиотек OWL и VCL.

Запуск IDE и выход

Для запуска интегрированной среды необходимо средствами операционной системы зайти в каталог Bin интегрированной среды и задать команду `BC` в командной строке, или щелчком левой кнопкой мыши на пиктограмме на рабочем столе (если вы позаботились о ее создании на рабочем столе). Сама интегрированная среда работает в консольном окне системы Windows, показанном на рис. П1.1.

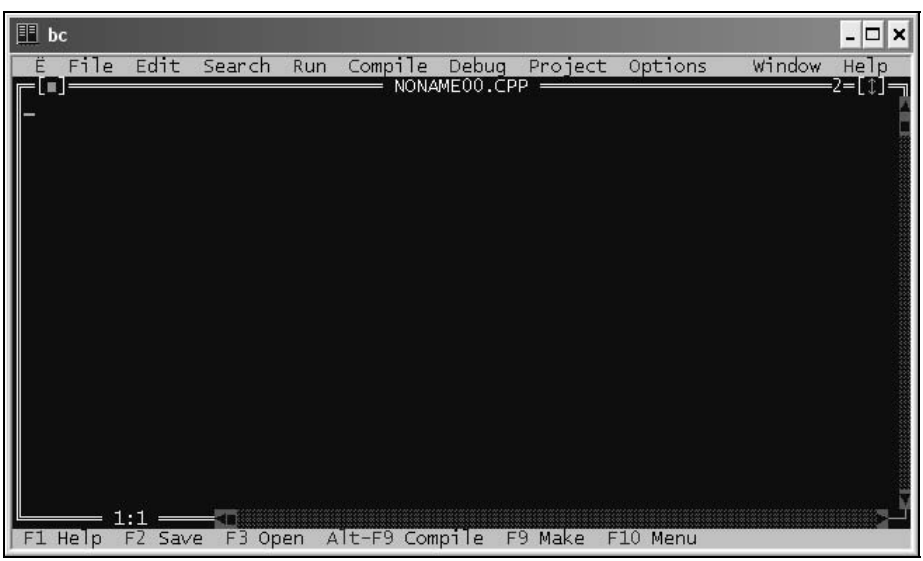

Рис. П1.1. Главное окно Borland C++ 3.1

Сразу открывается окно редактора и можно набирать программу. Назначение пунктов меню следующее:

- **?** — системное меню;
- **File** — управление файлами (создать, открыть, сохранить и т. п.);
- **Edit** — работа с буфером обмена (Clipboard);
- **Search** — поиск и замена;
- **Run** — различные режимы выполнения программы;
- **Compile** — компиляция и компоновка исполняемого модуля;
- **Debug** — управление возможностями отладчика;
- **Project** — управление проектами (многофайловых программ);
- **Options** — установка режимов работы интегрированной среды;
- **Window** — работа с окнами интегрированной среды;
- **Help** — обращение к справочной подсистеме.

Переход в меню выполняется клавишей <F10>. Переход в конкретный пункт меню — соответствующими "горячими клавишами" <Alt>+<Символ>.

Для того чтобы выйти из программы, есть три способа:

- через меню **File | Quit**;
- нажать комбинацию клавиш <Alt>+<F> и <Q>;
- нажать комбинацию клавиш <Alt>+<X>.

Создание и сохранение программы

Простая одномодульная программа (большинство программ-заданий для студентов являются таковыми) создается в окне редактора — для этого не требуется создавать проект. Для создания новой программы надо открыть новое окно редактора, что можно сделать двумя способами:

- через меню **File | New**;
- нажать комбинацию клавиш <Alt>+<F> и <N>.

На экране появится окно редактора синего цвета, как при первом запуске системы (см. рис. П1.1).

Набранную программу надо сохранить. Выполняем один из трех вариантов:

- через меню **File | Save**;
- нажать комбинацию клавиш <Alt>+<F> и <S>;
- нажать клавишу <F2>.

В консольном окне появится диалоговое окно, в котором система предлагает по умолчанию каталог (это каталог Bin интегрированной среды) и имя

nonamexx.cpp (на месте xx стоят две цифры.). Диалоговое окно представлено на рис. П1.2.

Мы должны задать имя и нажать клавишу <Enter>. Если нам потребуется сохранить программу с новым именем, то это делается посредством меню **File | Save As**, а диалоговое окно сохранения то же самое.

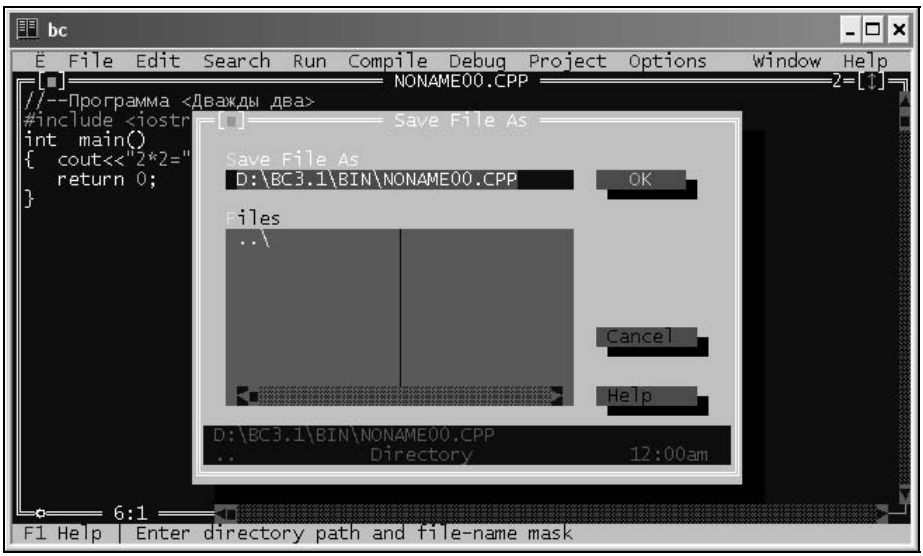

Рис. П1.2. Диалоговое окно для задания имени файла исходной программы

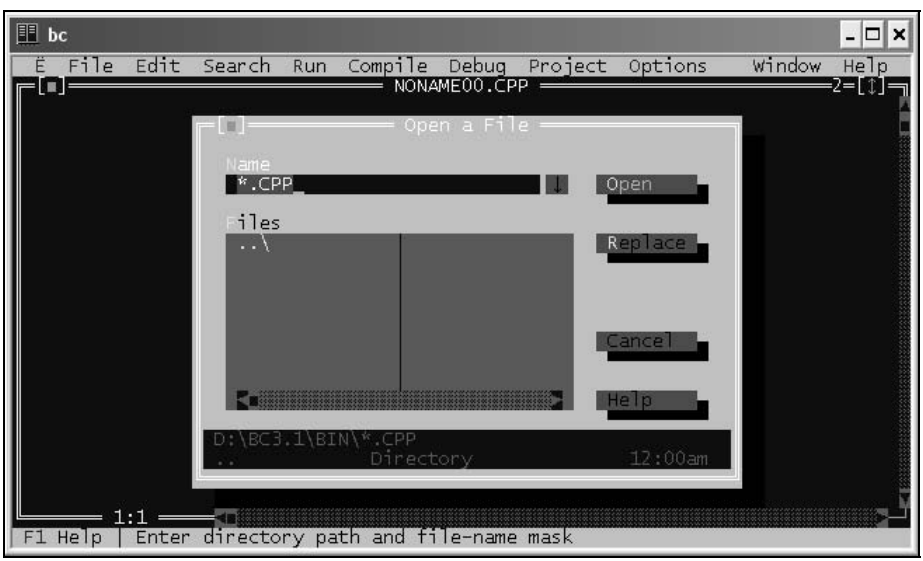

Рис. П1.3. Диалоговое окно открытия файла

Открыть файл с текстом программы можно так:

- через меню **File | Open**;
- нажать комбинацию клавиш <Alt>+<F> и <O>;
- нажать клавишу <F3>.

На экране появится диалоговое окно для выбора файла, показанное на рис. П1.3.

Нужно выбрать (или набрать) каталог и набрать имя файла. Расширение cpp можно не указывать.

Многофайловые программы

Системой поддерживается организация многофайловых проектов. Проект — это набор взаимосвязанных исходных файлов, и возможно, объектных файлов и библиотек, компиляция и компоновка которых приводит к созданию единственного исполняемого модуля. Меню **Project** содержит команды, необходимые для создания, изменения, открытия и закрытия проекта.

Создание и изменение проекта

Для создания проекта нужно выполнить следующие действия:

1. Выполнить команду **Project | Open project**. В окне, очень похожем на диалоговое окно открытия файла (см. рис. П1.3), нужно задать каталог и имя проекта.

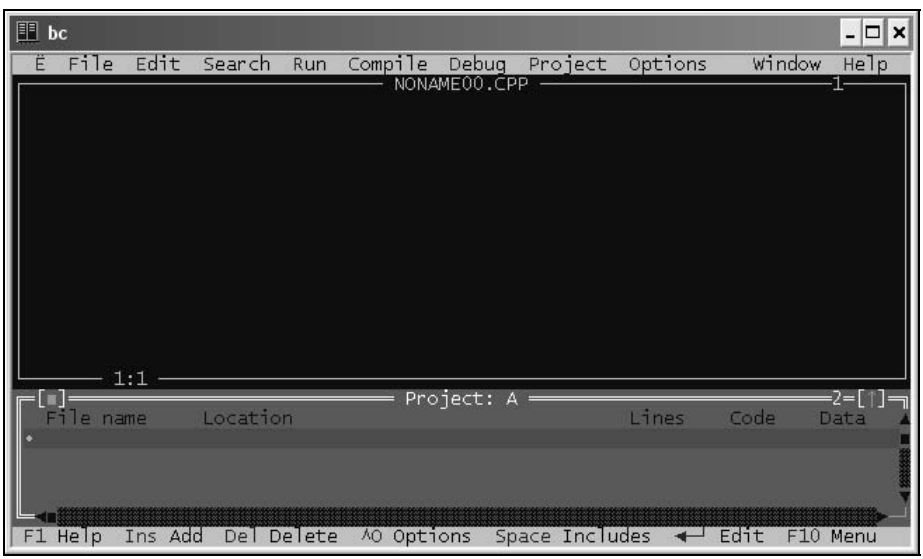

Рис. П1.4. Окно проекта

2. Добавить в окно проекта (рис. П1.4) те файлы, которые входят в проект (клавиша <Insert>, пункт меню **Project | Add Item**).

Удаление элементов проекта выполняется в открытом окне клавишей <Delete>, или с помощью меню **Project | Add Item**.

При работе с проектом выполнение команд меню **Compile | Build all** и **Compile | Make** отличается так: в первом случае транслируются все исходные файлы проекта, а во втором — только изменившиеся с момента последней трансляции.

Компиляция, компоновка и выполнение

Компиляция и компоновка программы выполняются с помощью меню **Compile**:

- **Compile** <Alt>+<F9> — проверка правильности синтаксиса;
- **Make** <F9> — создание исполняемого модуля;
- **Build all** — создание исполняемого модуля.

Разница между командами **Make** и **Build all** проявляется только в многомодульных программах.

Выполнение программы осуществляется с помощью меню **Run** (комбинация клавиш <Ctrl>+<F9>). При отсутствии ошибок и установленных точек прерывания программа будет выполняться от начала до конца. Результат выполнения программы можно увидеть либо в окне вывода интегрированной среды (меню **Window | Output**), либо в консольном окне (комбинация клавиш <Alt>+<F5>).

Можно задать аргументы командной строки (пункт меню **Run | Arguments**).

Работа с отладчиком

При наличии ошибок в программе мы можем воспользоваться возможностями интегрированного отладчика. Основное назначение отладчика — предоставить пользователю средства для управления процессом выполнения программы. Например, отладчик позволяет остановить и продолжить запуск программы, выполнить программу до определенного места, выполнить по операторам, просмотреть и изменить значения переменных программы.

В любой момент отладочное выполнение можно прервать (комбинация клавиш <Ctrl>+<F2>, пункт меню **Run | Program reset**).

Точки прерывания

Точка прерывания — это оператор, на котором следует остановить выполнение программы. Точка прерывания устанавливается и отменяется одной и

той же комбинацией клавиш <Ctrl>+<F8> (пункт меню **Debug | Toggle breakpoint**). Точка прерывания ставится на том операторе, где установлен курсор редактора. При установке точек прерывания программа будет выполняться до ближайшей точки, после чего остановится. Программист может отменить точку прерывания, продолжить выполнение программы, просмотреть значения переменных и т. п.

Выполнение до курсора

Курсор является специальной точкой прерывания. Выполнение программы можно остановить на операторе, на котором установлен курсор, если запустить выполнение комбинацией клавиш <Ctrl>+<F4> (или пункт меню **Run | Goto cursor**).

Пооператорное (пошаговое) выполнение

Выполнять программу по операторам можно либо сначала, либо после любой остановки программы. Пошаговое выполнение осуществляется в одном из двух режимов:

- с "заходом" в функции, определенные в программе (клавиша <F7>, пункт меню **Run | Trace into**).
- без "входа" в функции, при этом вызов функции считается неделимым оператором (клавиша <F8>, пункт меню **Run | Step over**).

Просмотр и изменение переменных

В момент остановки программы мы можем просмотреть переменные, объявленные на данном уровне вложенности. Во всех случаях, для просмотра необходимо в окне ввода задать имя переменной. Существуют различные варианты просмотра:

- просмотр в окне **Inspecting** (комбинация клавиш <Alt>+<F4>, пункт меню **Debug | Inspect**) (рис. П1.5);
- просмотр в окне **Evaluate and Modify** (комбинация клавиш <Ctrl>+ +<F4>, пункт меню **Debug | Evaluate and Modify**) (рис. П1.6);
- просмотр в окне **Watch** (комбинация клавиш <Ctrl>+<F7>, пункт меню **Debug | Watch**) (рис. П1.7).

Справочная система

Фирма Borland (в отличие, например, от Microsoft) во всех своих системах обязательно предоставляет пользователю прекрасную контекстно-зависимую справочную систему. Вызов контекстной справки обычно выполняется клавишей <F1>. Несмотря на "древность" системы, справка построена по

принципу гипертекста и позволяет перемещаться "по информации" привычным способом — по ссылкам в тексте. Через меню **Help** можно просто поискать нужную информацию. Меню содержит несколько пунктов:

- **Contents** — на экран вызывается главное окно справочной системы, в котором указано ее содержание (рис. П1.8);

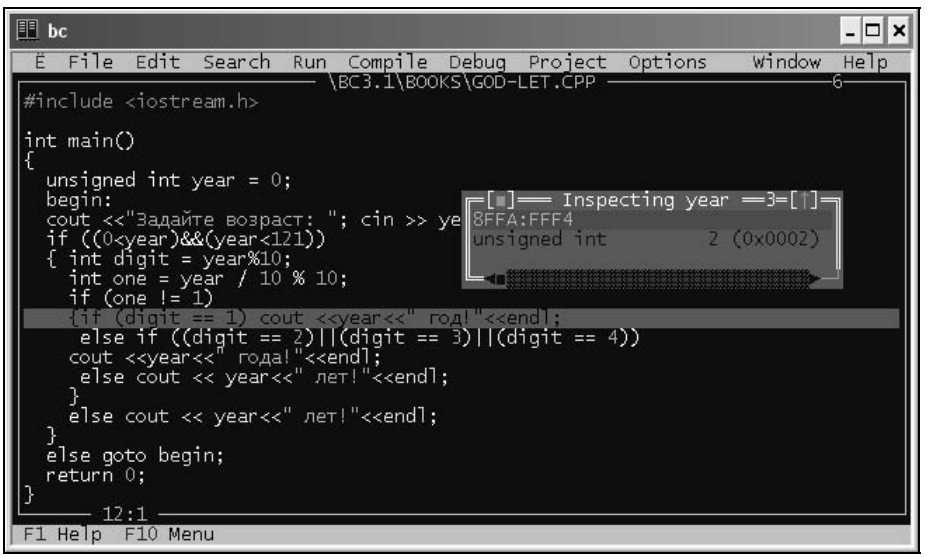

Рис. П1.5. Окно **Inspecting**

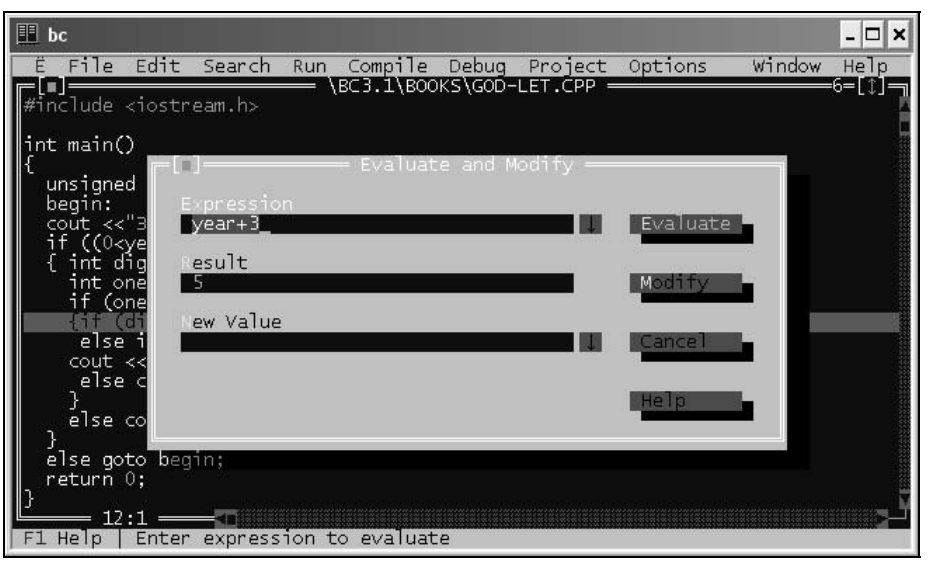

Рис. П1.6. Окно **Evaluate and Modify**

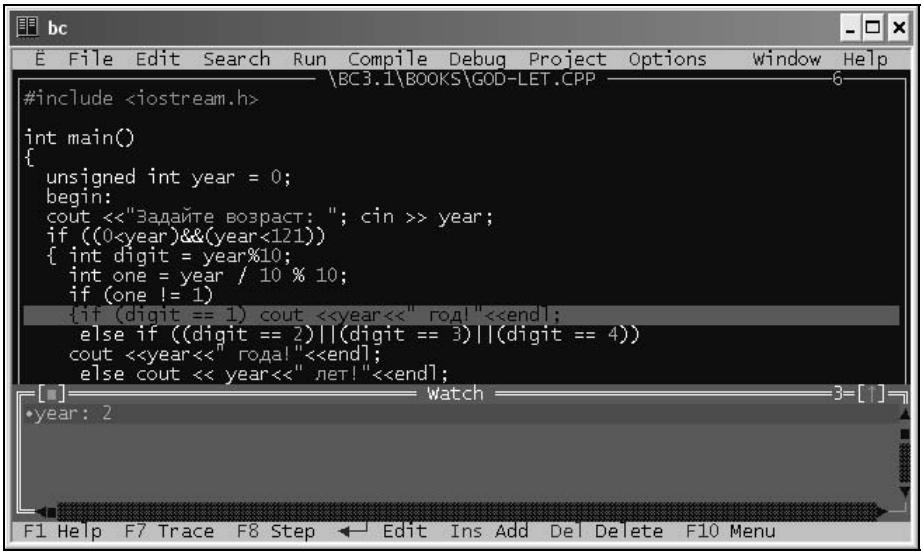

Рис. П1.7. Окно **Watch**

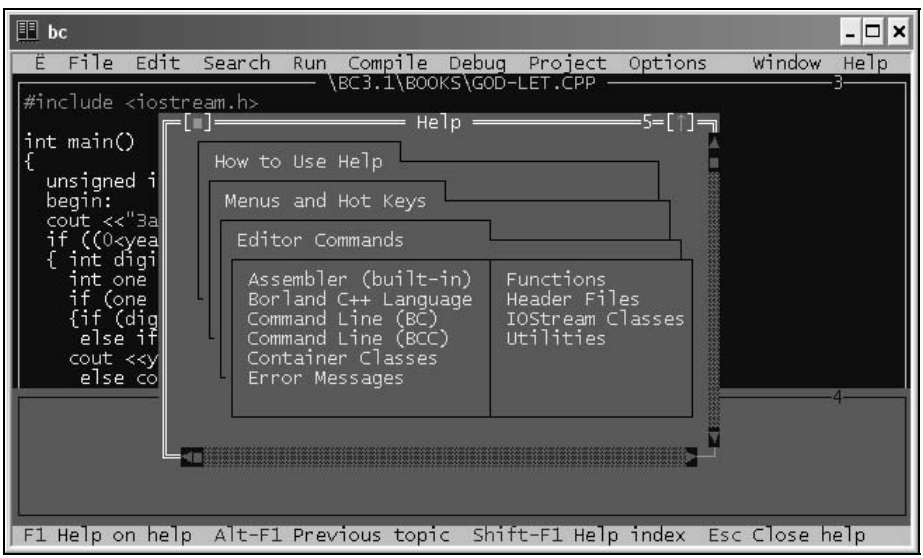

Рис. П1.8. Главное окно справочной системы

- **Index** — алфавитный индекс терминов (комбинация клавиш <Shift>+ +<F1>);
- **Topic search** — поиск нужного раздела справки (комбинация клавиш <Ctrl>+<F1>), позволяет получить справку о любой стандартной функции или конструкции C++ непосредственно из окна редактора: доста-

точно установить курсор на нужную конструкцию и нажать "горячие клавиши";

- **Previous topic** — показывает предыдущее окно справки.

Основными пунктами системы **Help** являются:

- **Editor Commands** — необходимые сведения о режимах работы и командах редактора интегрированной системы;
- **Borland C++ Language** — описание конструкций языка C++ и расширений, реализованных в системе;
- **Command Line (BC)** или **(BCC)** — описание работы двух компиляторов системы при их запуске из командной строки;
- **Functions** — описание всех функций, реализованных в системе;
- **Header Files** — описание всех заголовочных файлов, которые находятся в каталоге include;
- **Error Messages** — описание сообщений об ошибках.

Borland C++ 5

Эта система была создана непосредственно перед принятием стандарта, но включает почти все, что принято в стандарте C++. Поэтому основное ее назначение — создание приложений для Windows, хотя она обладает обратной совместимостью и позволяет создавать простые консольные приложения.

Запуск IDE

При установке на компьютер система записывается в главное меню Windows, поэтому вызов системы выполняется из главного меню. Вид окна после запуска представлен на рис. П1.9.

Как видим, главное меню отличается от системы Borland C++ 3.1 только тремя пунктами, но они для наших целей не понадобятся.

Выход из системы выполняется через меню **File | Exit**. Можно закончить работу и обычными "горячими клавишами" Windows <Alt>+<F4>.

Создание и выполнение консольных программ

Система Borland C++ 5 может работать с проектами и без проектов. Сначала рассмотрим более простой вариант: работа без проектов.

Для создания консольной программы достаточно выполнить пункт меню **File | New | Text Edit**.

Рис. П1.9. Главное окно Borland C++ 5

Откроется пока пустое окно редактора (рис. П1.10), в котором можно набирать текст программы. Обратите внимание, как изменился состав панелей управления.

В тексте программы, в самом конце, перед закрывающей скобкой нужно прописать вызов функции `getch()` из библиотеки conio.h. Это необходимо для того, чтобы увидеть результаты выполнения — при срабатывании этой функции программа остановится и будет "ждать" нажатия любой клавиши (после чего завершится). Библиотека conio.h не входит в стандарт, но реализована и в системах фирмы Borland, и в Visual C++ 6.

Набранную программу нужно сохранить. Выполняем один из вариантов:

- через меню **File | Save** или **File | Save As**;
- нажать комбинацию клавиш <Alt>+<F> и <S>;
- нажать комбинацию клавиш <Ctrl>+<S>.

В консольном окне появится диалоговое окно (рис. П1.11), в котором система предлагает по умолчанию каталог и имя nonamexx.cpp (на месте xx стоят две цифры).

Мы должны задать имя и нажать клавишу <Enter>. Если нам потребуется сохранить программу с новым именем, то это делается посредством меню **File | Save As**, диалоговое окно сохранения файла то же самое.

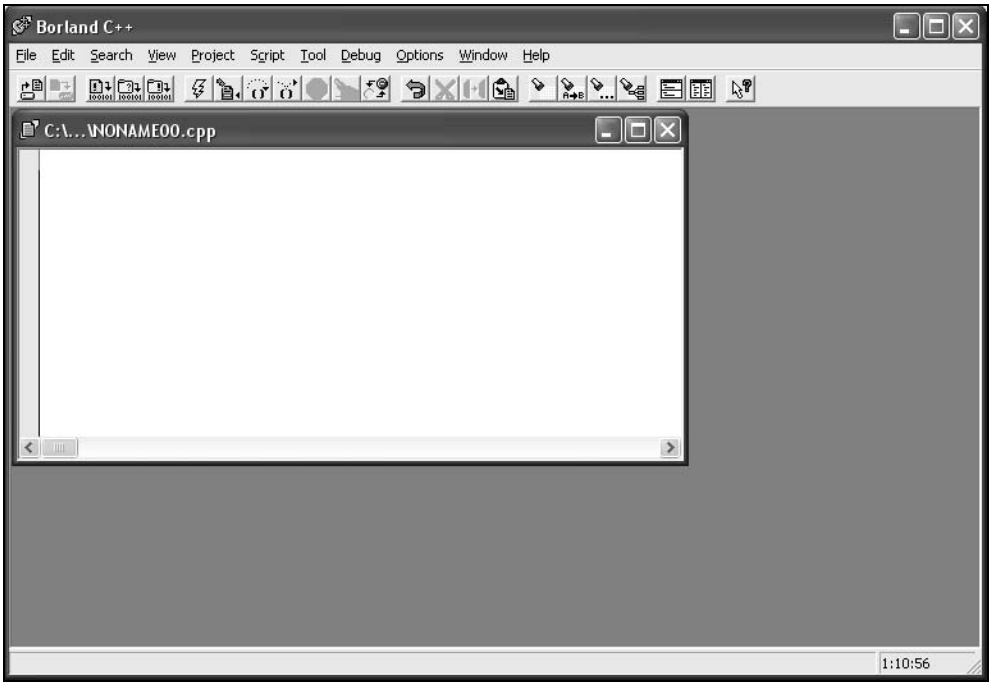

Рис. П.1.10. Открытое окно редактора

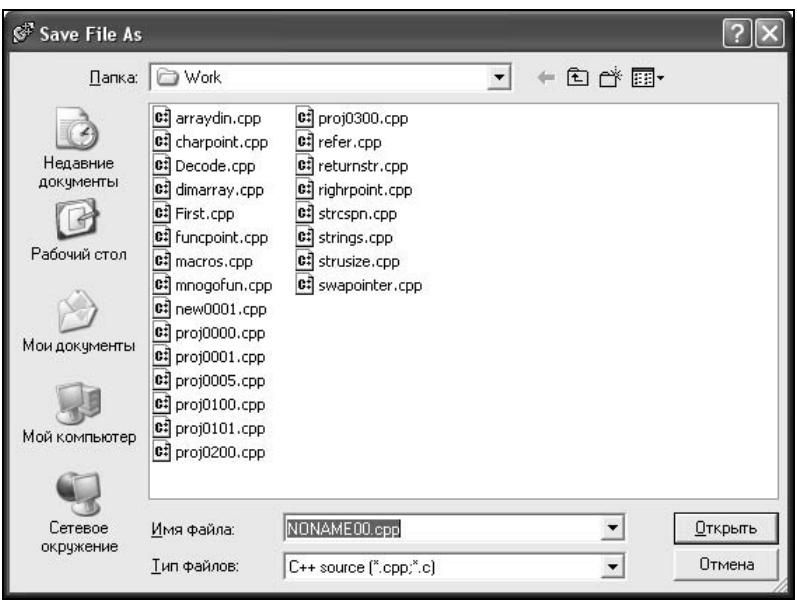

Рис. П1.11. Диалоговое окно задания имени файла исходной программы

Открыть файл с текстом программы можно следующими способами:

- через меню **File | Open**;
- нажать комбинацию клавиш <Alt>+<F> и <O>.

На экране появится диалоговое окно для выбора файла, показанное на рис. П1.12.

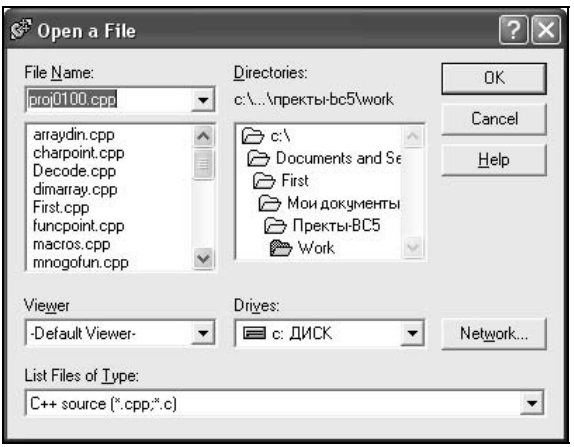

Рис. П1.12. Диалоговое окно открытия файла

Чтобы проверить, построить и выполнить программу, можно воспользоваться теми же "горячими клавишами" (<F9>, <Alt>+<F9>, <Ctrl>+<F9>), которые описаны ранее для системы Borland C++ 3.1. Но обычно для этой цели пользуются панелью инструментов, которая представлена на рис. П1.13.

Рис. П1.13. Панель инструментов для выполнения программы

Чтобы просто проверить и/или построить исполняемый модуль, надо использовать кнопки панели, которая показана на рис. П1.14.

Рис. П1.14. Панель инструментов для проверки модуля

Результаты система показывает в отдельном консольном окне, вид которого представлен на рис. П1.15.

Рис. П1.15. Окно результатов выполнения программы

Продолжение работы с программой

Чтобы закончить работу с текущей программой, надо просто закрыть окно.

Система "помнит" последнее рабочее состояние, поэтому при новом запуске автоматически открываются те окна, с которыми вы работали последний раз. Более того, в меню **File** записаны 9 последних файлов, с которыми вы работали. Если требуется продолжить работу с другой программой, то надо выполнить пункт меню **File | Open** и в открывшемся окне (см. рис. П1.12) выбрать нужный файл.

Создание проекта программы

Так как работа с программой без проекта не позволяет работать с отладчиком, то рассмотрим создание проекта программы. Для этого надо выполнить пункт меню **File | New | Project**.

В результате на экране откроется окно проекта, вид которого представлен на рис. П1.16. По умолчанию система предлагает каталог и имя проекта (поле **Project Path and Name**) по умолчанию. Имя исполняемого модуля прописывается автоматически в поле **Target Name**. Мы должны установить следующие параметры:

- в поле **Target Type** — Application [.exe];
- в поле **Platform** — Win32;
- в поле **Target Model** — Console.

Необходимо отключить все флажки и установить тип библиотеки **Static**, в поле **Libraries** — статическая компоновка с библиотечными модулями.

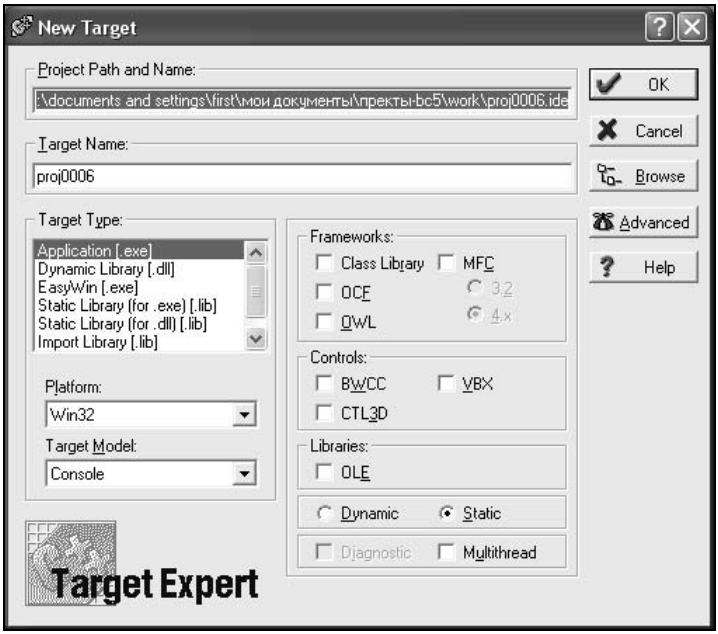

Рис. П1.16. Окно создания проекта

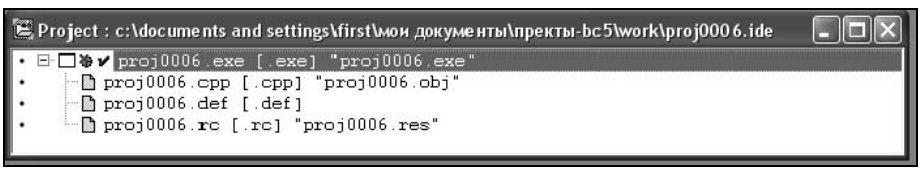

Рис. П1.17. Окно состава проекта

В результате откроется окно с составом проекта, представленное на рис. П1.17.

Для продолжения работы мы должны выполнить следующие действия:

1. Щелкнуть правой кнопкой мыши по последней строке и удалить узел (delete node).
2. Удалить предпоследнюю строку (результат удаления представлен на рис. П1.18).

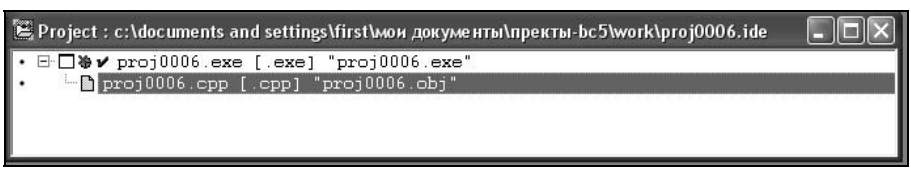

Рис. П1.18. Окно состава проекта после удаления узлов

3. Щелкнуть левой кнопкой мыши на имени исходного файла; в результате откроется окно редактора (см. рис. П1.10), в котором можно набирать текст программы.

Работа с отладчиком

Создание проекта программы позволяет провести ее отладку в интегрированной среде. Отладчик позволяет установить и отменить точки прерывания, выполнять программу по операторам с "заходом" в функции и без этого, выполнить программу до курсора. В момент остановки программы мы можем просмотреть и изменить значения переменных. Отладку в любой момент можно прервать и продолжить выполнение в непрерывном режиме. Все действия выполняются с помощью пунктов меню **Debug**. В этом меню указаны и "горячие клавиши", которые немного отличаются от клавиш системы Borland C++ 3.1, из-за того, что некоторые комбинации зарезервированы Windows.

Справочная система

Справку можно получить, выполнив пункт главного меню **Help**. В этом меню перечислены несколько пунктов, основными из которых являются:

- **Contents** — справка о возможностях интегрированной среды, в частности, подробное изложение возможностей отладчика;
- **Keyword search** (клавиша <F1>) — справка о работе с проектом. Клавиша <F1> выполняет контекстный поиск информации по ключевым словам и библиотечным функциям;
- **Windows API** — описаны функции, которые предоставляет Windows для программирования прикладных задач.

Справочная система имеет традиционный для Windows вид, представленный на рис. П1.19. На вкладке **Содержание** показаны разделы справки по возможностям интегрированной среды.

На вкладке **Предметный указатель** можно выполнять быстрый поиск информации по языку C++ и библиотечным функциям (рис. П1.20).

Вкладка **Поиск** позволяет выполнить более серьезный поиск информации в справочной системе. Если поиск осуществляется первый раз, то система сначала предлагает создать базу данных поиска, выводя окно мастера настройки поиска, представленное на рис. П1.21.

После создания базы данных появляется основное окно поиска, вид которого представлен на рис. П1.22.

Набирая в первой строке нужное слово (например, `vector`), в списке с номером **2** увидим варианты написания слова, а в списке с номером **3** появляются все темы справочной системы, в которых это слово встречается.

Рис. П1.19. Содержание справочной системы

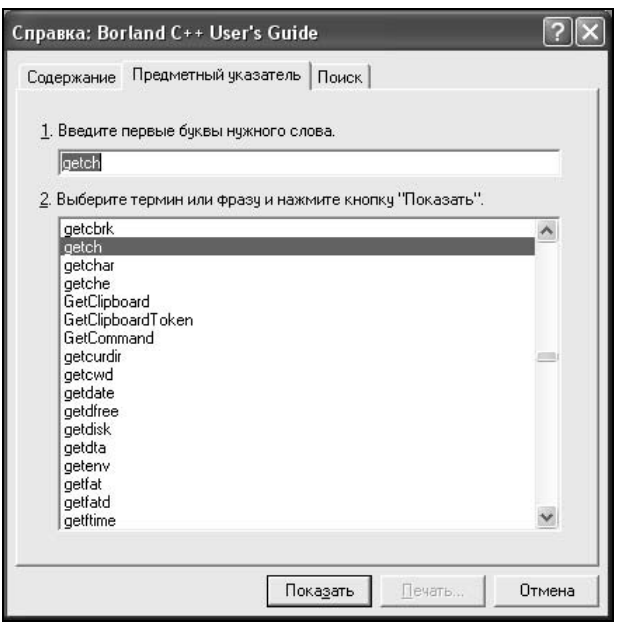

Рис. П1.20. Предметный указатель справочной системы

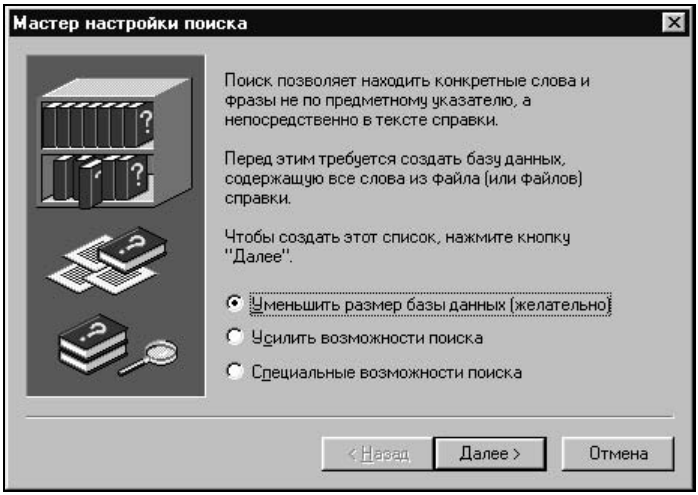

Рис. П1.21. Окно мастера настройки системы поиска

Рис. П1.22. Основное окно поиска в справочной системе

Borland C++ Builder 6

Система значительно новее, чем другие, рассматриваемые ранее системы — она создана в 2002 году. В связи с этим система значительно лучше поддерживает стандарт, чем предыдущие версии систем фирмы Borland, и лучше, чем система Visual C++ 6. Например, в качестве стандартной библиотеки шаблонов официально используется открытая библиотека STLport (адрес сайта приведен в *приложении 3*). Система предназначена для быстрой разработки приложений, однако позволяет программировать и простые консольные приложения, оконные приложения с использованием Windows API. Рассмотрим создание консольных приложений.

Запуск IDE

При установке на компьютер система записывается в главное меню Windows, поэтому ее вызов выполняется из главного меню. По умолчанию система сразу переходит в режим быстрой разработки приложений, отображая пустую форму *(см. гл. 14)*. Для создания консольного приложения необходимо отказаться от этого режима, выполнив пункт меню **File | Close All**.

Создание нового проекта

Для создания нового консольного приложения требуется выполнить пункт меню **File | New | Other**.

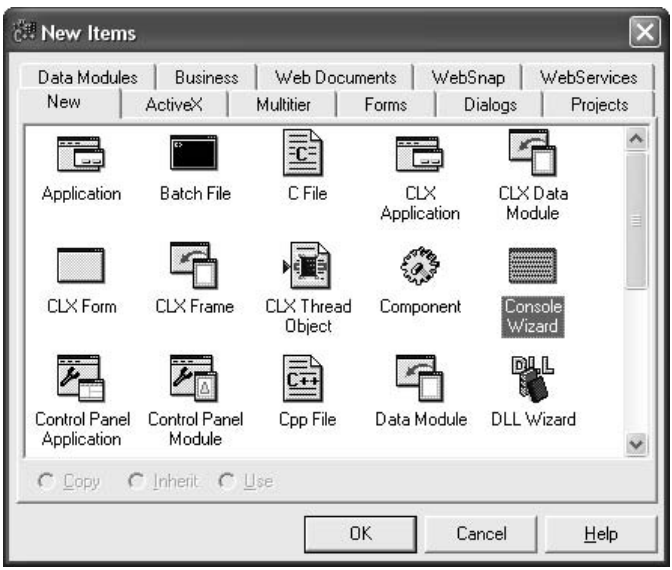

Рис. П1.23. Выбор типа приложения (консольное приложение)

На экране появится окно, содержащее типы приложений, которые можно создавать с помощью системы Borland C++ Builder 6. Внешний вид окна представлен на рис. П1.23.

Мы должны выбрать значок **Console Wizard** на вкладке **New** и нажать кнопку **OK**. Появится новое окно, вид которого представлен на рис. П1.24.

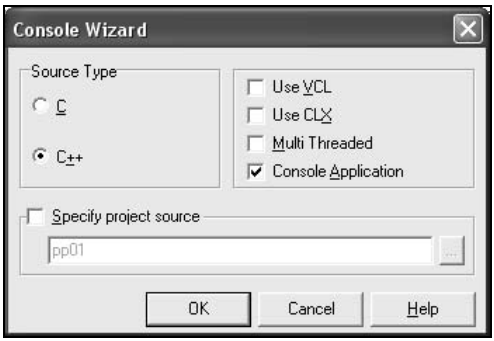

Рис. П1.24. Установка флажков Console Wizard

В этом окне мы должны установить флажок **Console Application** и задать язык программирования C++. После нажатия кнопки **OK** на экране появляется окно редактора, вид которого представлен на рис. П1.25.

В окне сразу прописывается заготовка консольной программы.

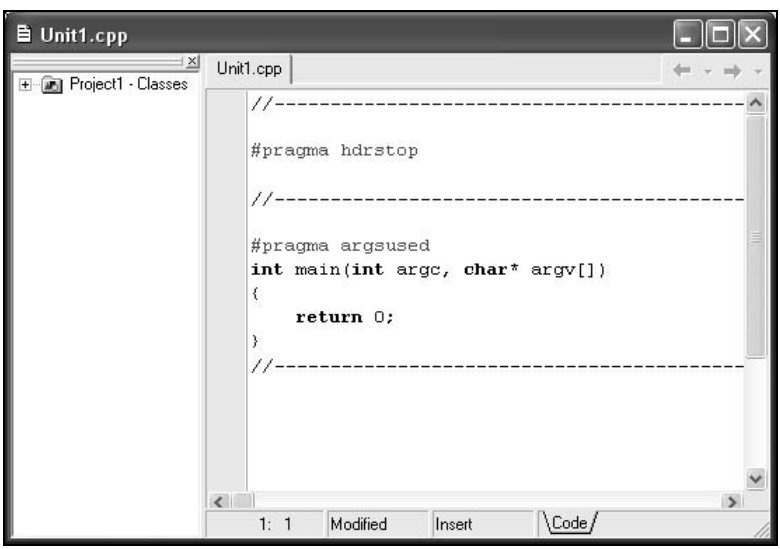

Рис. П1.25. Окно редактора с заготовкой консольной программы

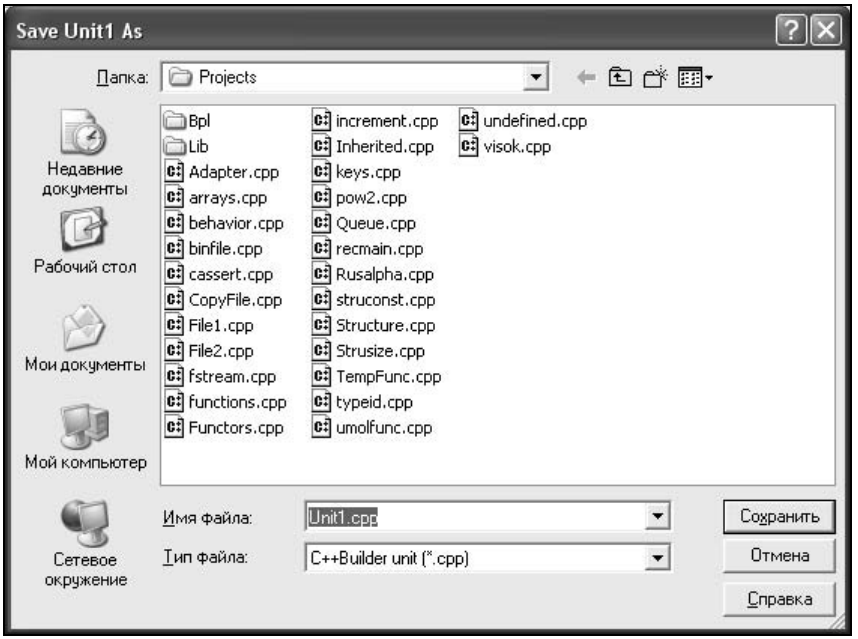

Рис. П1.26. Диалоговое окно сохранения файла исходной программы

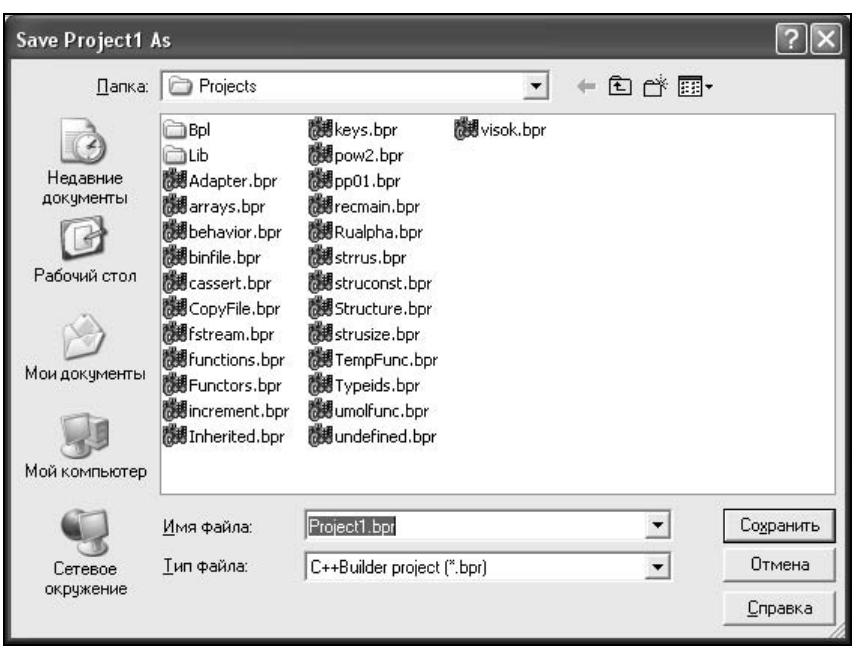

Рис. П1.27. Диалоговое окно сохранения файла проекта

Набранную программу надо сохранить. Выполняем один из вариантов:

- через меню **File | Save** или **File | Save As**;
- нажать комбинацию клавиш <Alt>+<F> и <S>;
- нажать комбинацию клавиш <Ctrl>+<S>.

В консольном окне появится диалоговое окно (рис. П1.26), в котором система предлагает по умолчанию каталог и имя файла unit1.cpp.

Мы должны задать имя файла и нажать кнопку **Сохранить**. Кроме того, при сохранении проекта (пункт меню **File | Save Project As**) или при закрытии проекта (пункт меню **File | Close All**) на экране появляется аналогичное диалоговое окно для сохранения проекта (вид окна представлен на рис. П1.27).

Продолжение работы с программой

Чтобы закончить работу с текущей программой, надо просто закрыть проект.

Система работает только с проектами (расширение файла bpr), поэтому при необходимости дальнейшей работы нужно открывать проект. Система "помнит" последние несколько проектов, с которыми работал программист — они записываются прямо в меню **File | Reopen**. Для проектов, которые были в работе относительно давно, надо выполнить пункт меню **File | Open Project** (комбинация клавиш <Ctrl>+<F11>). На экране появится окно, аналогичное диалогу сохранения проекта (см. рис. П1.27), только вместо кнопки **Сохранить** отображается кнопка **Открыть**.

Компиляция, компоновка и выполнение

Чтобы проверить/выполнить программу, можно воспользоваться теми же "горячими клавишами" (<F9>, <Alt>+<F9>, <Ctrl>+<F9>), которые описаны ранее для системы Borland C++ 3.1. Но обычно для этой цели пользуются панелью инструментов, которая представлена на рис. П1.28.

Рис. П1.28. Панель инструментов для выполнения программы

Работа с отладчиком

Работа с отладчиком ничем не отличается от работы в других системах фирмы Borland, поэтому подробно описывать ее нет необходимости. Все отладочные возможности доступны в меню **Run**.

Справочная система

Borland C++ Builder 6 имеет прекрасную справочную систему, которая, как обычно, является контекстно-зависимой. Однако при установке справочная система записывается отдельным пунктом в главное меню Windows, и может вызываться независимо от интегрированной среды. Меню помощи содержит 24 пункта, поэтому лучше создать пиктограмму на Рабочем столе и вызывать справочник в окне, как показано на рис. П1.29.

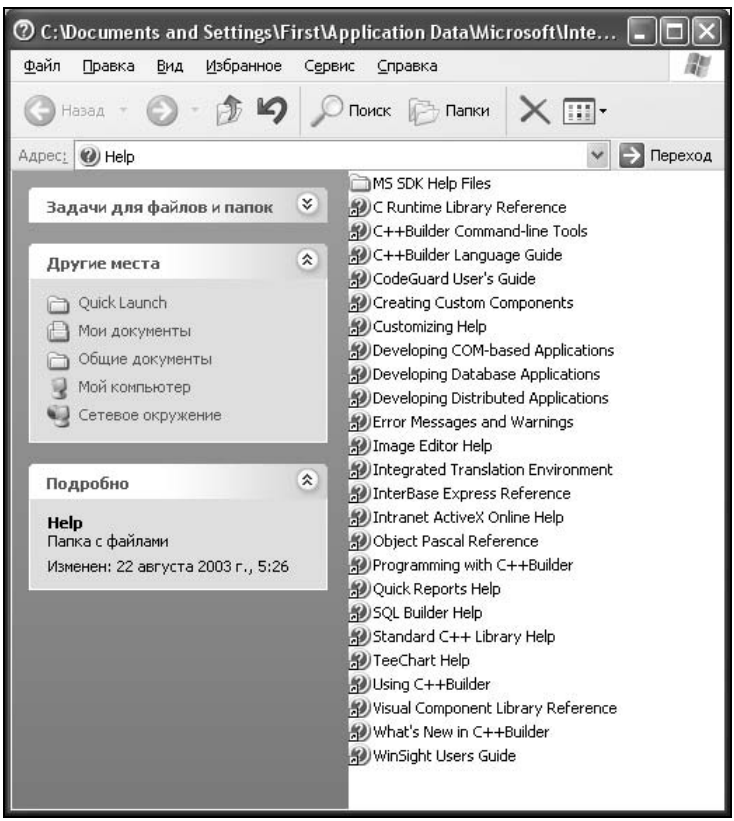

Рис. П1.29. Окно справочной системы

Выбирая нужный пункт, мы открываем окно справочной системы стандартного вида (см. рис. П1.19). Большинство разделов справки по языку C++ и стандартной библиотеке шаблонов взято непосредственно из стандарта C++.

ПРИЛОЖЕНИЕ 2

Интегрированная среда Microsoft Visual C++ 6

Система создана в 1998 году непосредственно перед принятием стандарта и реализует практически полный стандарт 98-го года. До сих пор эта система остается наиболее используемой для профессиональных разработок.

Запуск IDE

При установке система записывается в главное меню Windows, поэтому запускать ее необходимо через меню. Вид окна интегрированной среды представлен на рис. П2.1.

Окончание работы с системой выполняется либо с помощью меню **File | Exit** или стандартным для Windows способом — надо нажать комбинацию клавиш <Alt>+<F4>.

Создание и выполнение программ

Visual C++ 6 всегда работает с проектами. Самая простая программа, даже состоящая из трех строчек, всегда оформляется как проект. *Проект* — это набор взаимосвязанных исходных файлов, и возможно, объектных файлов и библиотек, компиляция и компоновка которых приводит к созданию единственного исполняемого модуля. Несколько проектов можно объединить в общее *рабочее пространство* — эта возможность предназначена для работы группы программистов над одним большим проектом. Для наших небольших примеров каждый проект будет создаваться в отдельном рабочем пространстве.

Для создания программы надо сначала создать проект. Для этого нужно выполнить приведенные далее действия.

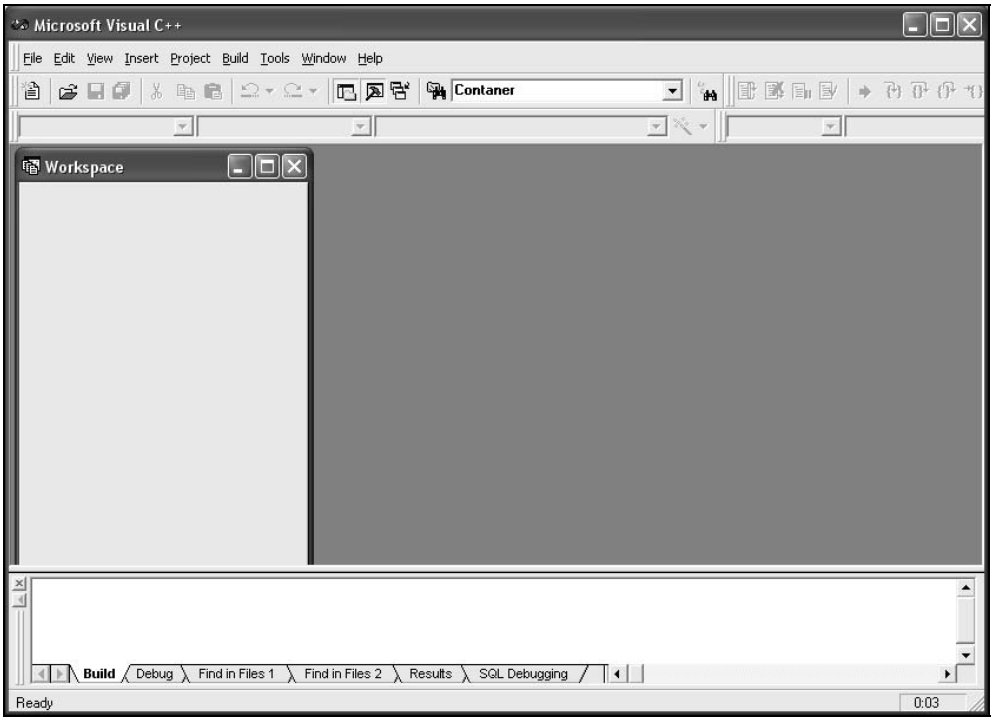

Рис. П2.1. Главное окно интегрированной среды

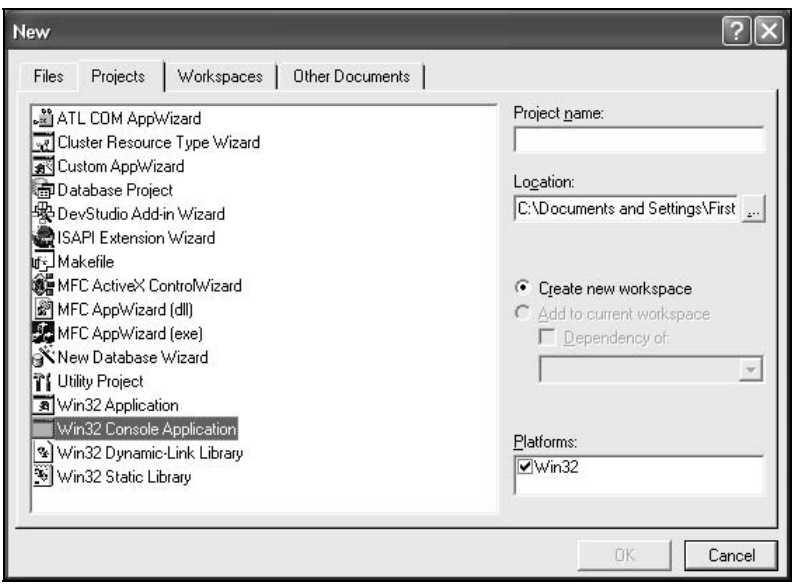

Рис. П2.2. Диалоговое окно **New**, вкладка **Projects**

1. Выбрать пункт меню **File | New**.
2. В появившемся диалоговом окне (рис. П2.2) выбрать вкладку **Projects** и пункт **Win32 Console Application**.
3. Набрать имя проекта в строке **Project name**, например `First`.

В результате в окне **Workspace** появляется дерево проекта, вид которого представлен на рис. П2.3.

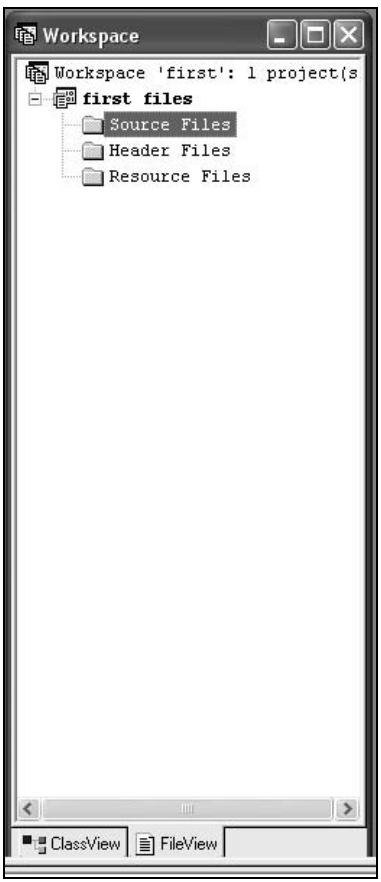

Рис. П2.3. Окно дерева проекта

Для каждого рабочего пространства (а в нашем случае имя пространства и имя проекта совпадают) система создает папку с указанным названием, внутри которой создает папку Debug и 5 служебных файлов. Все файлы имеют имя, совпадающее с именем папки, но разные расширения. Самый важный файл — это файл с расширением dsw. Файл объединяет информацию о проектах, входящих в рабочее пространство.

Теперь для создания исходного файла программы необходимо выполнить следующие действия.

1. Выбрать пункт меню **File | New** (см. рис. П2.1).
2. В появившемся окне (см. рис. П2.2) выбрать вкладку **Files** и пункт **C++ Source File**.
3. Набрать имя файла в строке **File names**, например `First`.

В появившемся окне редактора набирается текст программы (рис. П2.4).

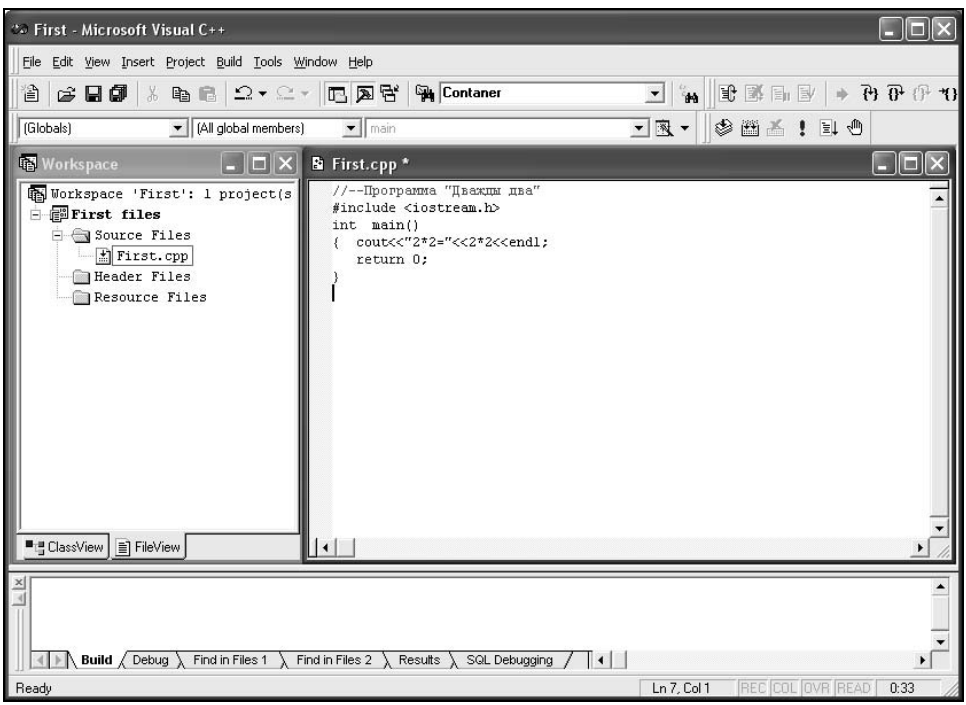

Рис. П2.4. Главное окно после ввода исходного текста программы

Чтобы выполнить программу, надо использовать панель инструментов **Build MiniBar**, которая представлена на рис. П2.5.

Результаты система показывает в отдельном консольном окне, вид которого представлен на рис. П2.6. Причем, в отличие от систем фирмы Borland сама приостанавливает выполнение, требуя нажатия любой клавиши (Press any key to continue).

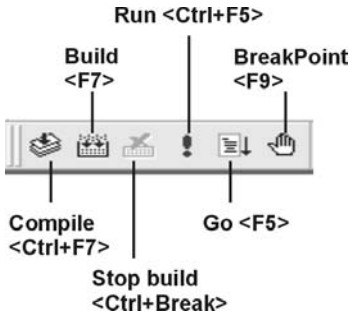

Рис. П2.5. Панель инструментов **Build MiniBar**

Рис. П2.6. Окно выполняемой программы

Продолжение работы с проектом

Чтобы закончить работу над проектом, надо закрыть рабочее пространство (пункт меню **File | Close Workspace**).

Чтобы продолжить работу над текущим проектом, необходимо выполнить пункт меню **File | Resent Workspace** и выбрать нужную строку. Если нам необходимо открыть проект, над которым мы давно не работали, то надо воспользоваться командой **File | Open Workspace**. В появившемся диалоговом окне (рис. П2.7) нужно выбрать файл с расширением dsw.

Чтобы добавить новый файл к проекту, надо просто создать новый исходный файл для уже открытого проекта. Файл автоматически включается в проект.

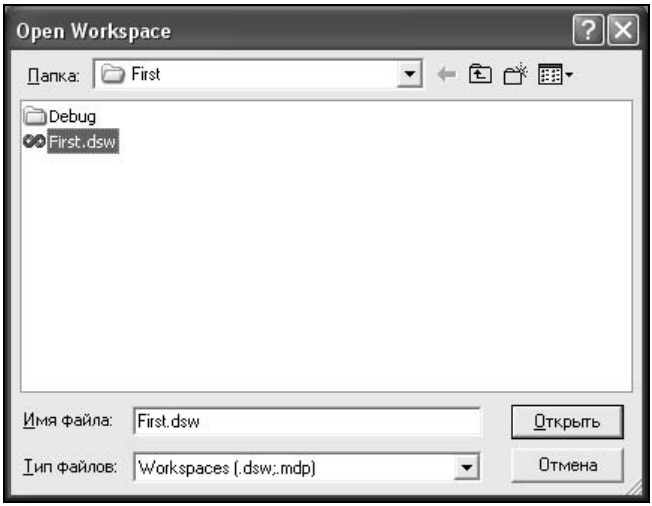

Рис. П2.7. Диалоговое окно открытия рабочего пространства

Конфигурация проекта

Visual C++ 6 создает исполняемый файл либо в отладочной конфигурации (Win32 Debug), либо в выпускной конфигурации (Win32 Release). По умолчанию принята отладочная конфигурация. Каждая конфигурация размещается в собственной папке внутри папки рабочего пространства (проекта). Для установки конфигурации необходимо выполнить пункт меню **Build | Set Active Configurations** и в появившемся диалоговом окне выбрать конфигурацию. Чтобы узнать текущую конфигурацию, надо выполнить пункт меню **Project | Settings**.

Работа с отладчиком

Работа с отладчиком выполняется либо с помощью меню **Debug**, либо используя панель управления **Debug**, вид которой представлен на рис. П2.8.

Рис. П2.8. Панель управления **Debug**

Отладчик системы Visual C++ 6 предоставляет все классические возможности отладки: установка точек прерывания, различные режимы выполнения программы, множество возможностей просмотра различной информации при выполнении программы.

Отладочное выполнение программы начинается при выполнении команды меню **Debug | Go** (или при нажатии соответствующей кнопки на панели управления). Вид экрана при отладке представлен на рис. П2.9.

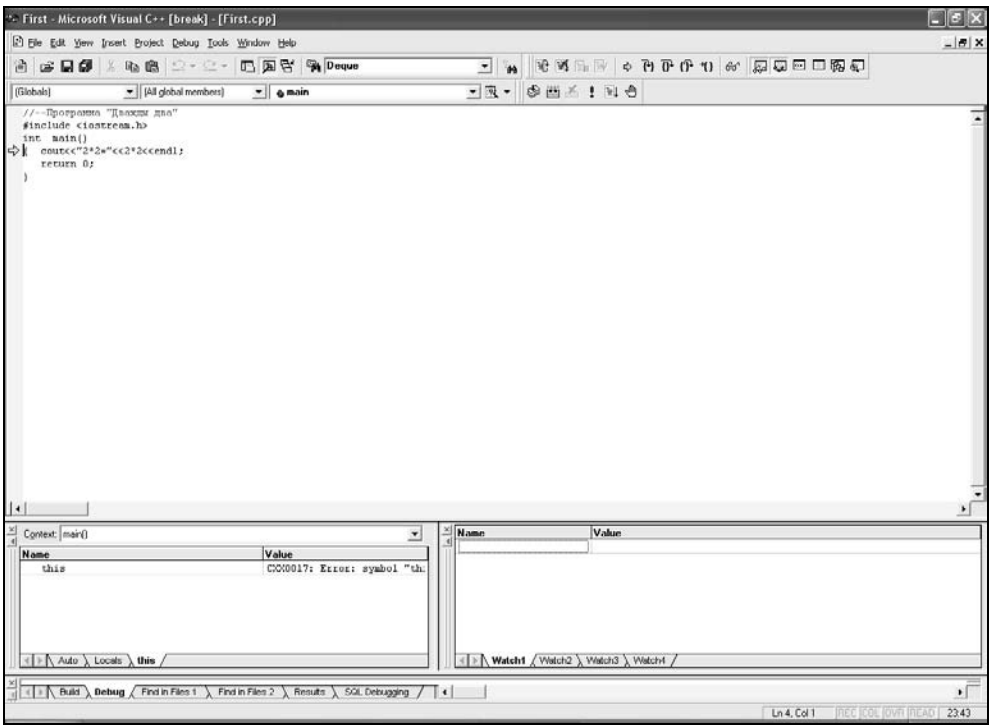

Рис. П2.9. Экран системы при отладке

Прекратить выполнение программы в отладочном режиме можно в любой момент, нажав кнопку **Stop Debugging** (или комбинацию клавиш <Shift>+ +<F5>).

ПРИЛОЖЕНИЕ 3

Ресурсы C++ в Интернете

- **http://www.research.att.com/~bs/homepage.html** — сайт автора языка C++, автора многих книг по C++, Бьярна Страуструпа.
- **http://anubis.dkuug.dk/jtc1/sc22/wg21/** — все о стандарте C++. Можно не только почитать стандарт, но и многочисленные предложения, поступающие в комитет по стандартизации.
- **http://msdn.microsoft.com** — сайт справочной информации компании Microsoft. Здесь есть все, но на английском языке.
- **http://rsdn.ru** — один из лучших сайтов для российских программистов. На сайте множество статей по нетривиальным вопросам программирования, есть раздел с хорошо написанными рецензиями на книги, собрано множество ссылок на программистские ресурсы Интернета. Небольшое сообщество квалифицированных специалистов поддерживает и развивает сайт, редактирует журнал RSDN. Но особенно сайт знаменит своими форумами. На мой взгляд, здесь лучшие программистские форумы на всем постсоветском пространстве.
- **http://clubpro.spb.ru/** — тоже неплохие форумы по разным вопросам C++, но уступают форумам сайта rsdn в оперативности.
- **www.cuj.com** — сайт самого популярного журнала по C++ "C++ users journal", где печатают статьи многие известные программисты, естественно, на английском языке.
- **http://www.gotw.ca/gotw/index.htm** — сайт Guru Of The Week, поддерживаемый Гербом Саттером. Именно здесь выставлены оригинальные формулировки задач, которые он потом изложил в своих книгах. В России издана объединенная книга [35].
- **http://www.aristeia.com/right_frames.html** — сайт Скотта Мейерса, чьи книги являются настольными для программистов на C++.

- **http://staff.develop.com/slip/** — Сайт Стэнли Липпмана, который тоже написал несколько хороших книг для изучающих C++.
- **http://www.firststeps.ru/** — сайт "Первые шаги", предназначенный исключительно для начинающих программистов.
- **http://www.cppwmeste.r2.ru/** — еще один сайт для начинающих "Изучаем вместе".
- Ряд сайтов являются фактически открытыми и бесплатными библиотеками документов о программировании:
  - **http://www.codetools.com/**
  - **http://www.codeproject.com/**
  - **http://www.codenet.ru/**
  - **http://www.codeguru.com/**
  - **http://www.doc.ic.ac.uk/lab/cplus/c++.rules/**
  - **http://www.helloworld.ru/**
  - **http://www.emanual.ru**
  - **http://www.sourcesl.ru**
  - **http://www.rusdoc.ru**
- Следующие три сайта в своей книге [25] подробно описывает Скотт Мейерс — эти сайты посвящены STL, причем оттуда можно скачать бесплатную версию этой стандартной библиотеки:
  - **http://www.sgi.com/tech/stl/** — сайт-учебник по STL;
  - **http://stlport.org/** — свободно распространяемая библиотека, которую, в частности, использует фирма Inprise в системе Borland C++ Borland 6;
  - **http://www.boost.org/** — самая "большая" свободная библиотека, включающая множество и стандартных, и не совсем обычных компонентов. Достаточно качественная документация.
- **http://gcc.gnu.org** — с этого сайта можно скачать свободно распространяемый довольно популярный компилятор C++.
- **http://www.digitalmars.com/** — еще один свободный компилятор и еще много всего.

Перечислить все сколько-нибудь значимые сайты Интернета, посвященные программированию, не представляется возможным. Поэтому еще обращаю ваше внимание на то, что практически любой печатный журнал имеет сайт в Интернете. Достаточно поискать с помощью любой поисковой системы, например Google.

ПРИЛОЖЕНИЕ 4

Люди перестают мыслить,
когда перестают читать.
Д. Дидро

Список литературы

1. Александреску А. Современное проектирование на C++. Серия C++ In-Depth, т. 3 / Пер. с англ. — М.: Вильямс, 2002. — 336 с., ил.
2. Аммерааль Л. STL для программистов на C++ / Пер. с англ. — М.: ДМК, 1999. — 240 с., ил.
3. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии, инструменты / Пер. с англ. — М.: Вильямс, 2001. — 768 с., ил.
4. Бадд Т. Объектно-ориентированное программирование в действии / Пер. с англ. — СПб.: Питер, 1997. — 464 с., ил.
5. Бентли Дж. Жемчужина программирования. 2-е издание / Пер. с англ. — СПб.: Питер, 2002. — 272 с., ил.
6. Берри Р. Микинз Б. Язык C: введение для программистов / Пер. с англ. — М.: Финансы и статистика, 1988. — 191 с., ил.
7. Браунси К. Основные концепции структур данных и реализация в C++ / Пер. с англ. — М.: Вильямс, 2002. — 320 с., ил.
8. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. / Пер. с англ. — М.: Бином; СПб.: Невский диалект, 1998. — 560 с., ил.
9. Вирт Н. Алгоритмы + структуры данных = программы / Пер. с англ. — М.: Мир, 1985. — 406 с., ил.
10. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Серия Библиотека программиста / Пер. с англ. — СПб.: Питер, 2001. — 368 с., ил.

11. Голуб А. И. С и C++. Правила программирования. — М.: Бином, 1996. — 272 с.

12. Гринзоу Л. Философия программирования для Windows 95/NT / Пер. с англ. — СПб.: Символ-Плюс, 1997. — 640 с., ил.

13. Дейтел П. Дж., Дейтел Х. М. Как программировать на C++. Введение в объектно-ориентированное проектирование с использованием UML / Пер. с англ. — М.: Бином, 2002. — 1152 с.

14. Каррано Ф. М., Причард Дж. Дж. Абстракция данных и решение задач на C++. Стены и зеркала, 3-е издание / Пер. с англ. — М.: Вильямс, 2003. — 848 с., ил.

15. Кениг Э., Му Б. Э. Эффективное программирование на C++. Серия C++ In-Depth, т. 2 / Пер. с англ. — М.: Вильямс, 2002. — 384 с., ил.

16. Керниган Б. В., Пайк Р. Практика программирования / Пер. с англ. — СПб.: Невский Диалект, 2001. — 381 с., ил.

17. Керниган Б., Ритчи Д. Язык программирования Си / Пер. с англ. 3-е изд., испр. — СПб.: Невский Диалект, 2001. — 352 с., ил.

18. Киммел П. и др. Borland C++ 5 / Пер. с англ. — СПб.: БХВ-Петербург, 2000. — 976 с., ил.

19. Круглински Д., Уингоу С., Шеферд Дж. Программирование на Microsoft Visual C++ 6.0 для профессионалов / Пер. с англ. — СПб.: Питер; М.: Русская Редакция, 2001. — 864 с., ил.

20. Липпман С. Б. Основы программирования на C++. Серия C++ In-Depth, т. 1 / Пер. с англ. — М.: Вильямс, 2002. — 256 с., ил.

21. Луис Д. Borland C++ 5. Справочник / Пер. с нем. — М.: Бином, 1997. — 560 с., ил.

22. Марченко А. Л. C++. Бархатный путь. — М.: Горячая линия — Телеком, 1999. — 400 с.

23. Мейерс С. Эффективное использование C++. 50 рекомендаций по улучшению наших программ и проектов. Серия Для программистов. / Пер. с англ. — М.: ДМК Пресс, 2000. — 240 с., ил.

24. Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению наших программ и проектов. Серия Для программистов / Пер. с англ. — М.: ДМК Пресс, 2000. — 304 с., ил.

25. Мейерс С. Эффективное использование STL. Библиотека программиста / Пер. с англ. — СПб.: Питер, 2002. — 224 с., ил.

26. Мешков А. В., Тихомиров Ю. В. Visual C++ и MFC / Пер. с англ. — СПб.: БХВ-Петербург, 2000. — 1040 с., ил.

27. Москвин П. В. Азбука STL. — М.: Горячая линия — Телеком, 2003. — 262 с., ил.

28. Олафсен Ю., Скрайбер К., Уайт К. Д. и др. MFC и Visual C++ 6. Энциклопедия программиста / Пер. с англ. — СПб.: ДиаСофтЮП, 2003. — 992 с.

29. Павловская Т. А. C/C++. Программирование на языке высокого уровня. — СПб.: Питер, 2002. — 464 с., ил.

30. Павловская Т. А., Щупак Ю. А. C/C++. Структурное программирование: Практикум. — СПб.: Питер, 2002. — 240 с., ил.

31. Подбельский В. В. Язык C++. — М.: Финансы и статистика, 1996. — 560 с., ил.

32. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация / Под ред. Матросова. — СПб.: Питер, 2002. — 688 с., ил.

33. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ. — СПб.: Питер; М.: Русская Редакция, 2001. — 752 с., ил.

34. Романов Е. Л. Язык Си++ в задачах, вопросах и ответах. Серия Учебники НГТУ. — Новосибирск: НГТУ, 2003. — 428 с.

35. Саттер Г. Решение сложных задач на C++. Серия C++ In-Depth, т. 4 / Пер. с англ. — М.: Вильямс, 2002. — 400 с., ил.

36. Себеста Р. Основные концепции языков программирования, 5-е изд. / Пер. с англ. — М.: Вильямс, 2001. — 672 с., ил.

37. Страуструп Б. Язык программирования C++, спец. изд. / Пер. с англ. — М.: Бином; СПб.: Невский Диалект, 2001. — 1099 с., ил.

38. Страуструп Б. Дизайн и эволюция C++. Серия Для программистов / Пер. с англ. — М.: ДМК Пресс, 2000. — 448 с., ил.

39. Сэджвик Р. Фундаментальные алгоритмы на C++. Анализ. Структуры данных. Сортировка. Поиск / Пер. с англ. — К.: ДиаСофт, 2001. — 688 с.

40. Сэджвик Р. Фундаментальные алгоритмы на C++. Алгоритмы на графах / Пер. с англ. — СПб.: ДиаСофтЮП, 2002. — 496 с.

41. Фридман А., Кландер Л., Михаэлис М., Шилдт Х. C/C++. Архив программ. — М.: Бином, 2001. — 640 с., ил.

42. Халперн П. Стандартная библиотека C++ на примерах / Пер. с англ. — М.: Вильямс, 2001. — 336 с., ил.

43. Харт Дж. М. Системное программирование в среде Win32 / Пер. с англ. — М.: Вильямс, 2001. — 464 с., ил.

44. Хенкеманс Д., Ли М. Программирование на C++ / Пер. с англ. — СПб.: Символ-Плюс, 2002. — 416 с., ил.

45. Хэзфилд Р., Кирби Л. и др. Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений. Энциклопедия программиста / Пер. с англ. — К.: ДиаСофт, 2001. — 736 с.

46. Холзнер С. Visual C++ 6: учебный курс — СПб.: Питер, 1999. — 576 с., ил.

47. Шалыто А. А., Туккель Н. И., Шамгунов Н. Н. Ханойские башни и автоматы. — Программист, 2002, № 8.

48. Шилдт Г. Теория практика C++ / Пер. с англ. — СПб.: БХВ-Петербург, 2000. — 416 с., ил.

49. Элджер Дж. C++: библиотека программиста. — СПб.: Питер, 1999. — 320 с., ил.

50. Юань Фень. Программирование графики для Windows. — СПб.: Питер, 2002. — 1072 с., ил.

51. Programming languages — C++. International Standart ISO/IEC 14882:1998(E).

Предметный указатель

А

Адаптер:
◊ итератора 207, 408
◊ контейнера 195, 408
Адрес:
◊ байта 152
◊ константы 149
◊ массива 152
◊ отсутствие 148
◊ памяти 145, 147
◊ переменной 145, 148, 154, 177
◊ последнего параметра 265
◊ указателя 149
Адресное выражение 263
Алгоритм 67
◊ возвращающий итератор 212
◊ доступа к параметрам 265
◊ итеративный 274
◊ копирования 224
◊ модифицирующий 209, 216
◊ не модифицирующий 210
◊ поиска 210
◊ рекурсивный 272, 398
◊ сортировки 210, 214
◊ сортировки и поиска 212
◊ стандартный 61, 176, 195, 210, 334, 401
◊ типовой 195
◊ универсальный 210
◊ частичной сортировки 215
Алиас:
◊ внутреннего пространства имен 382
◊ имени 381
Альтернатива 27
◊ по умолчанию 28
◊ совпавшая 28
Анонимный объект 310
◊ динамический 312
Аргумент:
◊ главной функции 181
◊ массив 115
◊ фактический 267
◊ шаблона 412
Ассоциативный контейнер:
◊ множество 195
◊ мультимножество 195
◊ мультисловарь 195
◊ словарь 195
Атрибут доступа 377

Б

Байт 35, 47
◊ нулевой 122, 226
Безопасность типов 370
Бесконечный цикл 33
Библиотека 67
◊ ctype.h 95, 97
◊ math.h 65
◊ stdarg.h 270
◊ stdio 221
◊ stdlib.h 65, 100, 156
◊ STL 62, 214
◊ STLport 222
◊ string.h 135, 159, 188
◊ ввода/вывода 221, 241

Продолжение рубрики см. на с. 500

Библиотека *(прод.)*:
◊ интегрированной среды 62
◊ контейнеров 183
◊ стандартная 128
◊ стандартных программ 61
◊ шаблонов 62, 176, 183, 196, 340
Блок 26, 67
◊ контролируемый 349
◊ обработки исключений 349, 351
Буфер ввода/вывода 225
Буферизация потока 225

В

Ввод:
◊ значений 21, 55
◊ информации 221
◊ символа 224
◊ символьных массивов 225
◊ строк 127, 225
◊ чисел 55, 225
◊ числовых данных 223
Ввод/вывод:
◊ байтовый 251
◊ консольный 423
◊ символьный 251
◊ форматирование 257
Вектор:
◊ динамический 203
◊ количество элементов 203
◊ параметр 199, 239
◊ переменного размера 198
◊ по ссылке 199
◊ строк 202
Венгерская нотация 417, 418, 419
Видимость элементов класса 323
Виртуальная функция 409
Возврат:
◊ в поток 253
◊ итератора 212
◊ контейнера 201
◊ рекурсивный 281
◊ ссылки 259, 260
◊ указателя 262
◊ указателя на функцию 398
Время жизни:
◊ указателей 173
◊ переменной 77
Вставка элементов 171
Вывод:
◊ в файл 252
◊ информации 221
◊ массива 125
◊ на экран 22, 56
◊ приглашения 34, 55
◊ программы 47
◊ результатов 19
◊ строк 125
Вызов:
◊ вложенность 281
◊ деструктора 367
◊ конструктора 310
◊ конструктора неявный 307
◊ косвенный 395
◊ метода производного класса 372
◊ неоднозначность 332
◊ по типу объекта 371
◊ по типу указателя 371
◊ по указателю 389
◊ по умолчанию 405
◊ по условию 276
◊ рекурсивный 275—277, 290, 296, 365
◊ слева от присваивания 261
◊ функции 74—76, 401
  ▫ слева 212
◊ функциональный 298
◊ через указатель базового класса 372
Выражение 20, 24, 34, 36, 49, 50—52, 54—56, 59, 60, 63, 71, 73, 124
◊ адресное 154
◊ индекс массива 107
◊ константное 94, 106, 113
◊ логическое 24, 43, 53
◊ переключающее 27
◊ порядок вычисления 57, 59
◊ целочисленное 43
Выражение константное 411
Вычисление константных выражений 411

Г

Группа элементов управления 444

Д

Датчик случайных чисел 205
Двоичное дерево 171
Двумерный массив символов 162
Деление, целое 110
Деструктор 344
◊ автоматический 331
◊ базового класса 367, 409

◊ виртуальный 373
◊ класса 330, 342
◊ по умолчанию 331
◊ порядок вызова 367
Динамическая переменная:
◊ контейнер 203
Динамические структуры данных 171
Динамический массив указателей 167
Директива:
◊ pragma 141
◊ using 188
Длина строки 126
Доступ:
◊ двойной косвенный 267, 268
◊ дисциплина 171, 195
◊ к полям структуры 136
◊ к списку параметров 265, 270
◊ к элементам вектора 203
◊ к элементам класса 323
◊ к элементам контейнера 187, 195
◊ к элементам пространства имен 380
◊ к элементам структуры 129, 178
◊ косвенный 165, 264, 408
◊ модификатор 362
◊ по индексу 184
◊ по итератору 203
◊ по ключу 196
◊ последовательный 184, 195
◊ прямой 249

Е

Единица трансляции 381
◊ файл 377

З

Завершающий нуль 126
Завершение аварийное 351
Заголовок 158
◊ включаемого файла 188
◊ главной функции 179
◊ окна 428
◊ стандартный 191
◊ функции 264
◊ функции-операции 296
Запись файла 249
Зарезервированное слово 19, 21, 23, 30, 36, 38, 43, 63, 101, 184, 314, 323, 347, 373
Значение:
◊ "истина" 37
◊ "ложь" 37
◊ возвращаемое 19
◊ логическое 37, 95
◊ начальное 38
◊ переменной 21
◊ по умолчанию 74, 76
◊ фактического параметра 73
◊ цифры 124

И

Идентификатор 21, 62, 70
◊ пункта меню 432
◊ сообщения 429
Иерархия:
◊ исключений 358
◊ классов 357
◊ наследования 403
◊ стандартных исключений 358
Имя 21, 50
◊ включаемого файла 18
◊ главной функции 19
◊ каталога 229
◊ локальное в модуле 378
◊ макроса 100
◊ массива, указатель 166
◊ область видимости 69
◊ объекта-исключения 356
◊ параметра 181
◊ переменной 21
◊ ссылки 146
◊ типа 128
◊ файла 228
◊ функции 19, 62
◊ элемента управления 445
Индекс:
◊ конечный 116
◊ начальный 116
Инициализация:
◊ вектора 197
◊ в списке инициализации конструктора 311
◊ динамического контейнера 203
◊ динамической переменной 164
◊ массива 107
◊ многомерного массива 119
◊ полей 128
◊ поля-константы 311

Продолжение рубрики см. на с. 502

Инициализация *(прод.)*:
◊ символьный массив 186
◊ статической переменной 80
◊ строк 185, 190
◊ указателя на функцию 388
Инкапсуляция 68, 77, 87, 204, 361, 401
Инкремент:
◊ постфиксный 37, 50, 56
◊ префиксный 37, 50
Интегрированная среда 21
Интерфейс 409
◊ графического устройства 439
◊ класса 327, 329
◊ прикладного программирования 417
◊ программы 430
Информация о типе 374
Исключение 375
◊ динамический объект 353
◊ обработка 348
◊ объект 348
◊ стандартное 357, 359
Итератор 184, 187, 195
◊ ввода 224
◊ виды 206
◊ вставки 207, 208
◊ вывода 208
◊ действительный 209
◊ категории 206
◊ конечный 206
◊ константный 206
◊ начальный 206
◊ недействительный 209
◊ неконстантный 206
◊ обратный 205
◊ объект-посредник 204
◊ полуоткрытый интервал 206
◊ потоковый 208
◊ произвольный 207
◊ прямой 205

К

Каталог 228
◊ вложенный 229
◊ главный 231
◊ родительский 229
◊ стандартный 190
◊ текущий 229
Квалификатор имени компонента 380
Класс 322
◊ абстрактный 409
◊ базовый 362, 363, 366
◊ динамический 347
◊ дочерний 365
◊ интерфейс 324, 370
◊ исключений 354, 358
◊ методы 324
◊ окна 427
◊ параметризованный 387
◊ полиморфный 374
◊ потомок 361
◊ предок 361
◊ производный 362, 363
◊ родитель 363
◊ родительский 365
◊ статические элементы 408
◊ строк 184
◊ функциональный 403
Кнопка:
◊ элемент управления 452
Кодировка:
◊ символов 49, 158, 420
◊ универсальная 420
Количество элементов:
◊ массива 106
◊ не задано 107
◊ параметр 109
Комментарий 17
◊ многострочный 18
Компонент:
◊ базовый класс 450
◊ визуальный 444
◊ не визуальный 445
Компоновщик 377
Конец файла 224
Конечный автомат 398
◊ распознающий 398
Константа 34, 71
◊ "истина" 37, 43
◊ "ложь" 37
◊ в альтернативе 28
◊ восьмеричная 41
◊ десятичная 41
◊ дробная 36, 59
◊ имя массива 152
◊ математическая 39
◊ объявление 38
◊ определенная в Windows 230
◊ параметр шаблона 138
◊ символьная 47
◊ статическая 412
◊ строка 122, 124

◊ строковая 20, 22, 46
◊ суффикс 41
◊ текстовая 30
◊ указатель 149
◊ целая 38, 42
◊ шестнадцатеричная 41
Конструктор 131, 365
◊ базового класса 365, 366
◊ без аргументов 341
◊ и преобразование типов 312
◊ инициализации 164, 307, 366, 403
◊ класса 330
◊ копирования 307, 308
◊ наследника 365
◊ по умолчанию 307, 364
◊ порядок вызова 367
◊ список инициализации 310
◊ структуры 306
◊ тело 366
◊ тривиальный 346
◊ формы 452
Контейнер 105, 195, 407
◊ STL 238
◊ ассоциативный 172, 195
◊ библиотеки STL 169
◊ дек 340
◊ динамический 196
◊ неупорядоченный 211
◊ последовательный 195, 200, 340
◊ символов 187
◊ словарь 339
◊ список 171, 204
◊ стандартный 339
Контекст устройства 439
Контроль типов 370
Копирование разрушающее 345
Корень дерева 173
Косвенное обращение 143, 145, 206
◊ двойное 149

M

Макет:
◊ ввода 224, 236
◊ информации 240
Макрос 100, 411
◊ NULL 149
◊ встроенный 101
◊ отладочный 103
◊ с параметрами 100
◊ стандартный 102
Манипулятор 20, 47, 213
◊ boolalpha 184, 241
◊ endl 233
◊ освобождения буфера 244
◊ ширины поля ввода 241
Маска флагов 241
Массив 105
◊ ассоциативный 196
◊ байтов 144
◊ внутри структуры 134
◊ входной 114
◊ выходной 114
◊ двумерный 119
◊ динамический 165, 330, 332
◊ и структура 134
◊ из одного элемента 107
◊ индекс 107
◊ инициализация 107, 121
◊ количество измерений 119
◊ количество элементов 106
◊ константный 114
◊ массивов 121
◊ многомерный 119, 166
◊ обнуление 107
◊ объявление 106
◊ параметр 109
◊ переменной длины 187
◊ размер 107
◊ с задаваемыми границами 327
◊ символов 121, 123, 157, 190
◊ символьный 123
◊ строк 121
◊ структур 134
◊ указателей 163, 167, 168
◊ указателей на строки 162
◊ указателей на функцию 389
◊ чисел 109
◊ числительных 191
Меню:
◊ выпадающее 431
◊ главное 431
◊ команда 432
◊ контекстное 431
◊ приложения 446
◊ пункты 431
◊ системное 431, 442
Метавычисление 387, 411
Метапрограммирование 410
Метод вне шаблона 138

Механизм:
◊ RTTI 359
◊ декомпозиции 376
◊ исключений 347, 354
◊ обработки исключений 350
◊ перегрузки 93
Многократное присваивание 50
Множество, контейнер 201
Модель памяти 151
Модификатор:
◊ доступа 362
◊ наследования 362
Модуль стандартный 377
Мультиметоды 403

Н

Наследник абстрактного класса 409
Наследование 357, 361, 365
◊ закрытое 363, 368
◊ защищенное 363
◊ иерархия 361
◊ множественное 409
◊ открытое 363, 368
◊ простое 361
Неопределенное поведение 60, 88
Неэффективность рекурсивной функции 278, 281
Номер элемента массива 152

О

Область видимости 379, 380, 381, 391
◊ имени 77
◊ локальная 260
◊ наименьшая 382
◊ потоковой переменной 228
◊ указателя 164
Область действия переменной 82
Обобщенное программирование 387
Обработка:
◊ исключения 349
◊ ошибок 332
Обход дерева 287
Объединение 177, 268
Объект:
◊ анонимный 337
◊ базового класса 366
◊ временный 193, 307
◊ динамический 343
◊ исключение 348, 349
◊ отрицатель 406
◊ привязка 406
◊ производного класса 366
◊ текущий 317, 319, 321
◊ функциональный 215, 402, 403
Объявление:
◊ константы 38, 42
◊ объединения 177
◊ переменной 21, 26
◊ с инициализацией 161
◊ согласование 377
◊ указателя 148
◊ указателя на функцию 390
Окно:
◊ выпадающее 438
◊ главное 433
◊ дочернее 433
◊ модальное 451
◊ рабочая область 428, 439
Оконная процедура 425, 433
Оператор 19—21, 23, 28, 30, 32, 50, 51, 55, 57, 58, 60
◊ typedef 43, 112
◊ ввода 20, 26, 225
◊ возврата 19, 31, 67
◊ вывода 19, 23, 34, 46, 225
◊ выхода из блока 28, 33, 34
◊ генерации исключения 348
◊ метка 39, 45
◊ обработки исключения 357
◊ переключатель 27, 28, 244, 396, 429, 436
◊ перехвата исключения 349
◊ перехода 39, 45
◊ прекращения текущей итерации 34
◊ преобразования типа 52
◊ составной 23, 26, 30, 31, 32
◊ условный 23, 25, 28, 46, 54
◊ цикла 30, 31, 32, 33, 54, 55, 106
  ▫ итерация 29, 32, 54
  ▫ с постусловием 31
  ▫ с предусловием 30
  ▫ со счетчиком 32
  ▫ тело 30, 31, 56
Операция 20, 34, 59
◊ delete 164
◊ new 164
◊ sizeof 36
◊ арифметическая 37, 51
◊ бинарная 56, 317

◊ битовая 145, 250
◊ ввода 223
◊ ввода/вывода 235
◊ встроенная 298
◊ вывода 223
◊ вызов функции 56
◊ вызова виртуальная 403
◊ декремента 37
◊ деления целых 44
◊ доступа 203
◊ запятая 55, 56, 59, 72
◊ И 24, 46, 53
◊ ИЛИ 46, 53
◊ индексирования 57, 198, 328
◊ инкремента 37, 152
◊ логическая 46, 53, 60
◊ логическое отрицание 53
◊ многократная 51
◊ над элементами массива 115
◊ обмена 235
◊ одноместная 319
◊ получения адреса 145
◊ получения остатка от деления 44
◊ постфиксная 319
◊ префиксная 319
◊ приоритет 57
◊ присваивания 37, 49, 71, 72, 328, 332, 366
◊ присвоить 54
◊ равенства 54
◊ разрешения контекста 79, 137, 365, 382
◊ разыменования 146, 164, 206, 344
◊ с присвоением 51
◊ с символами 49
◊ сокращенная 53
◊ сравнения 37, 44
◊ сцепления 187
◊ унарная 53, 329
◊ условная 52
◊ чтения 235, 253
Определение:
◊ рекурсивное 280, 411
◊ функции 280
Отрицатель 407
Ошибка:
◊ выполнения 59
◊ грубая 113
◊ код 348
◊ компоновки 379
◊ набора 236
◊ распространенная 232
◊ серьезная 111
◊ синтаксическая 113
◊ случайная 107
◊ трансляции 59, 186, 263, 366
◊ утечка памяти 165, 327
Ошибки трансляции 101

П

Память:
◊ возврат 165
◊ выделение 167
◊ динамическая 175
◊ утечка 165, 166
Парадигма программирования 387
Параметр:
◊ в указательной форме 148
◊ динамический массив 166
◊ количество элементов 110
◊ контейнер 199
◊ массив 109
◊ начальное значение 289
◊ обязательный 394
◊ передаваемый по ссылке 260
◊ предикат 405
◊ следующий 265
◊ ссылка 263
◊ ссылка на массив 112
◊ указатель 144, 263
◊ указатель на функцию 392
◊ функция 176
◊ явный 264, 270
Параметры:
◊ алгоритмов 204
◊ в рекурсивных функциях 289
◊ главной функции 426
◊ командной строки 179
◊ константные 84
◊ массивы 144
◊ по умолчанию 72, 73, 91
▫ в прототипе 79
◊ список 62, 67
◊ строки 189
◊ тип 71
◊ фактические 63
◊ формальные 63, 70
◊ функции main 179
◊ шаблона 94, 343
Паттерн 407
◊ адаптер 408
◊ проектирования 204, 407

Перегрузка 157
◊ внешними функциями 301
◊ возведения в степень 315
◊ вызова функции 337
◊ деструкторов 330
◊ и спецификация исключений 354
◊ индексирования 339
◊ инкремента 299
◊ конструкторов 307
◊ неоднозначность 90
◊ операции вывода 300
◊ операции вызова функции 402
◊ операции сравнения 305
◊ операции умножения 303
◊ операций 294
◊ операций функциями-методами 316
◊ сложения с присваиванием 301
◊ функций и операций 361
◊ шаблонов 94
Передача указателя на функцию 392
Передача массива по указателю 110, 113
Передача параметра:
◊ по значению 70, 85
◊ по ссылке 70, 82, 84, 297
◊ по указателю 70
Передача указателя:
◊ по ссылке 154
◊ по указателю 154
Перекодировка строк 227
Переменная 21, 24, 26—28, 30, 31, 34, 38, 44, 50, 51, 54, 56, 59, 60, 64, 71, 423
◊ глобальная 77, 78, 87, 379
◊ динамическая 164, 170, 174, 327
◊ значение 21, 22, 29, 31
◊ имя 21
◊ инициализация 31, 38
◊ класса окна 426
◊ локальная 26, 67, 77, 78, 84, 278, 364
◊ не определена 82
◊ нелокальная 86
◊ необъявленная 81
◊ объявление 21
◊ символьная 48
◊ среды 181
◊ статическая 80, 281
◊ счетчик 29, 32
◊ тип 21, 58
◊ указатель 145
◊ целая 146
Перемещение:
◊ к концу файла 250
◊ к началу файла 250
Переопределение метода 364
Переполнение 57
◊ стека 278
Побочный эффект 290
Позиция:
◊ потока 249
◊ текущая 249
Поиск:
◊ в массиве 111
◊ минимального элемента 211
◊ минимального элемента массива 261
Полиморфизм 89, 361
◊ времени выполнения 374
◊ времени компиляции 374
◊ динамический 374
◊ статический 374
Поля защищенные 366
Последовательность:
◊ элементов контейнера 204, 212
◊ элементов массива 175
Последовательный контейнер:
◊ вектор 195
◊ дек 195
◊ очередь 195
◊ приоритетная очередь 195
◊ список 195
◊ стек 195
Поток:
◊ входной 221, 222, 232
◊ выходной 221, 222
◊ двунаправленный 222
◊ последовательный 249
◊ связанный с файлом 227
◊ символов 223
◊ создание 228
◊ состояние 234
◊ стандартный 222
◊ строковый 222, 257
◊ уничтожение 228
◊ файловый 222
Предикат:
◊ бинарный 407
◊ унарный 404
Преобразование:
◊ автоматическое 368
◊ неявное 344
◊ при вводе/выводе 223, 248
◊ типа 394
▫ указателя 267

◊ типов 401
  ▫ неявное 314
◊ указателей 150, 248, 401
◊ явное 297
Преобразование типа 51, 148
◊ динамическое 374
◊ запретить 314
◊ неявное 50
◊ перекрестное 375
◊ повышающее 375
◊ понижающее 375
◊ потеря информации 52
◊ явное 52, 150
Препроцессор 18, 99
◊ директивы 18, 99
◊ макросы 100
◊ тело макроса 100
◊ условная компиляция 99
Префикс 356
◊ имени 188, 418, 419, 429
◊ константы 234
Приведение типов 395
◊ явное 450
Привязка 407
Признак конца файла 253
Приложение:
◊ консольное 179, 421
◊ оконное 179, 425
Принцип инкапсуляции 309, 323, 330
Присваивание:
◊ векторов 198
◊ срезка 367
◊ структур 129
Проверка данных 236
Программа рекурсивная 274
Пространство имен 377
◊ анонимное 383
◊ глобальное 382
◊ локальное 383
◊ объявление 380
◊ полное 188
◊ состав 380
◊ стандартное 188, 191, 380
Прототип 76
◊ бинарной операции 295
◊ локальный 80
◊ область видимости 79
◊ унарной операции 295
◊ функции 157
◊ функции-операции 295
◊ функции-параметра 175

P

Разделитель стандартный 239
Размер:
◊ булевского типа 184
◊ динамического массива 166
◊ массива 126
◊ объединения 178
◊ стека 278
Разность итераторов 207
Расширение имени файла 229
Режим:
◊ компилятора 141
◊ открытия файла 246
Режимы работы системы ввода/вывода 241
Рекурсивная форма 272
Рекурсивный алгоритм 288
Рекурсивный возврат 283, 284
Рекурсивный подъем 284
Рекурсивный спуск 283, 284
Рекурсия 272
◊ бесконечная 276
◊ в шаблонах 411
◊ глубина 276, 278, 284
◊ косвенная 273
◊ прямая 273
◊ условие окончания 274
◊ условие продолжения 274

C

Свойства:
◊ компонента 445
◊ настройка 446
Связанные структуры 171
Селектор 164
Символ 420, 447
◊ двухбайтный 421
◊ завершения ввода 227
◊ завершитель строки 227
◊ заполнитель 242
◊ значащий 226
◊ конца строки 251
◊ недопустимый 223
◊ новой строки 251
◊ однобайтный 421
◊ разделитель 223, 229
◊ удалить из потока 227
Система тестирования 201

Системный объект 19, 21, 222
Ситуация:
◊ аварийная 347
◊ исключительная 348, 350, 354
◊ конца файла 357
◊ ошибочная 347
Смещение записи 249
События 447
◊ обработчик 447
Сообщение:
◊ о завершении программы 104
◊ об ошибке 92, 115, 332, 348
◊ обработка 436
◊ при выполнении 111
◊ при трансляции 111
Сортировка:
◊ массива 117, 329
◊ по убыванию 218
◊ шаблон 117
Состояние:
◊ нормальное 235
◊ ошибки потока 235
◊ потока 235
◊ тяжелое 235
Специализация шаблона:
◊ полная 413
◊ частичная 413
Список:
◊ двусвязный 171
◊ инициализации 365
◊ исключений 353
◊ кольцевой 171
◊ линейный 171, 284
◊ односвязный 171
◊ параметров 264
◊ связанный 340
◊ списков 172
◊ указателей 269
Список параметров 89
◊ переменной длины 72, 394
  ▫ макросы 270
◊ фактических 72
◊ формальных 72
◊ шаблона 93
Способ передачи параметров 265, 267
Ссылка 343
◊ висячая 174
◊ на локальную переменную 259
◊ на статическую переменную 260
◊ на указатель 175
◊ потерянная 174, 332
Стандарт C++ 43, 60, 70, 83, 88, 94, 150, 159, 179, 180, 183
Стандартная библиотека 18, 183, 195, 405
◊ версии 186
◊ ввода/вывода 221
Стандартные алгоритмы и контейнеры 204
Стандартный функциональный объект 405
Стандартный каталог 191
Стандартный поток 240, 244
◊ ввода 222
◊ вывода 222
Строка 123
◊ символьная 123
Структура 105, 127
◊ ввод 129
◊ вывод 129
◊ и указатели 169
◊ инициализация 128
◊ как параметр 129
◊ параметр 130
◊ параметр по умолчанию 131
◊ поля 128
◊ размер 140
◊ с указателями 169
◊ связанные 171
◊ селектор 129
◊ тег 128
◊ шаблон 132
Схема:
◊ пошаговая 280
◊ программы 23, 25, 29, 238
◊ работы ввода/вывода 225
◊ реализации 201
◊ рекурсивная 273, 279
◊ функции 273
◊ функции корректировки 256

T

Текущий диск 233
Текущий каталог 233
Тип 19, 21, 36, 52, 56
◊ аргументов 70, 265
◊ базовый для указателя 266
◊ булевский 184
◊ возвращаемого значения 19, 63, 67, 89
◊ возвращаемый 288
◊ встроенный 21, 46

◊ данных 35, 43, 53
◊ динамической переменной 164
◊ динамической структуры 173
◊ исключения 348, 350, 352
◊ контейнера 204
◊ логический 38, 43, 184
◊ модификатор 42
◊ нестандартный 191
◊ неявное преобразование 24
◊ параметризация 387
◊ параметров 158, 332
◊ первого параметра 267
◊ переменная 21, 58
◊ перечислимый 42
◊ перечислимый в структуре 137
◊ полная спецификация 42
◊ преобразование 49, 51
◊ размер 35
◊ символьный 46, 47
◊ символьный универсальный 421
◊ строковый 46, 121
◊ указателя на указатель 395
◊ указателя на функцию 396
◊ файла 229
◊ формального параметра 161
◊ целочисленный 43
◊ целый 19, 42, 106
◊ числовой 21
◊ явное преобразование 48
Типовой прием 204
Точность:
◊ повышенная 64
◊ чисел 242
Трансляция раздельная 377

У

Удаление элементов 171
Узел:
◊ дерева 286
◊ дочерний 173
◊ лист 173
◊ списка 171
Указатели 143
◊ арифметика 152
◊ виды 147
◊ и контейнеры 203
◊ инициализация 148
◊ объявление 148
◊ параметры 153
◊ передаются по значению 153
◊ разность 153
◊ элементы контейнера 203
Указатель 187
◊ "длинный" 419
◊ бестиповый 147, 369
◊ возвращаемое значение 157
◊ двойной 395
◊ значение по умолчанию 157
◊ имя функции 176
◊ интеллектуальный 165, 343
◊ константа 151
◊ константный 149
◊ локальный 175
◊ на абстрактный класс 409
◊ на базовый тип 371
◊ на контейнер 203
◊ на массив указателей 179
◊ на производный тип 371
◊ на следующий параметр 270
◊ на структуру 169
◊ на текущий объект 317
◊ на узел 287, 288
◊ на указатель 149, 167, 169, 175, 394
◊ на функцию 147, 388
◊ неявный 162
◊ нулевой 180
◊ переменная 145
◊ размер 151
◊ статический 389
◊ типизированный 147, 268, 395
Умножение матриц 166
Условие 24
◊ в операторе цикла 31
◊ в условном операторе 23, 34
◊ продолжения 287
Утечка динамической памяти 174

Ф

Файл 227
◊ двоичный 248, 249
◊ закрытие 228
◊ настроек 242
◊ открытие 228
◊ полное имя 229
◊ путь 229
◊ связывание с потоком 228
◊ текстовый 247
Файловая система 221, 228
Файловый поток 228, 231, 232

Флаг:
◊ состояния 235
◊ форматирования 241
Флажок, элемент управления 453
Форма:
◊ автоматически создаваемая 452
◊ главная 444
◊ приложения 443
◊ рекурсивной функции 276, 287
Форма записи:
◊ инфиксная 295, 317
◊ функциональная 295
Форма инкремента:
◊ постфиксная 299
◊ префиксная 299
Функтор 388, 402, 404
Функциональный объект 217
◊ арифметический 218
◊ стандартный 217
◊ универсальный 407
Функция 18, 62, 409, 414
◊ API 417, 428
◊ API Windows 229
◊ inline 331
◊ main 19, 68, 179
◊ атрибут 377
◊ бесконечная 277
◊ ввода символов 96
◊ виртуальная 373, 374
◊ возвращающая итератор 214, 262
◊ возвращающая ссылку 259
◊ возвращающая указатель 158, 262, 264
◊ вывода в файл 251
◊ вывода матрицы 168
◊ вывода строки 251
◊ вызов 63, 67
◊ вычисления длины строки 157
◊ вычисления максимума 67
◊ вычисления площади треугольника 352
◊ вычисления факториала 272
◊ генерации случайных чисел 65
◊ главная 18, 378, 425, 428
◊ глобальная 377
◊ двоичного поиска 116
◊ завершения 350
◊ заглушка 246, 252
◊ заголовок 19, 67, 69
◊ записи 246
◊ имя 67
◊ инициализирующая 131
◊ как параметр 175
◊ квадратный корень 64
◊ кодирования 251
◊ компонентная 136
◊ консольная 425
◊ константная 334
◊ локальная в модуле 378
◊ математическая 63
◊ метод 136
◊ не возвращающая значение 63
◊ не генерирующая исключений 353
◊ НОД 301
◊ обмена 333
◊ обработки строк 123, 135, 162
◊ обхода списка 285
◊ объект 402
◊ объявление 63
◊ окна 425, 427, 429, 432, 439
◊ определение 69
◊ отображения окна 428
◊ очистки экрана 244
◊ параметризованная 387
◊ параметры 70
◊ перегрузка 79, 89
◊ перекодировки 190, 240
◊ побочный эффект 85, 88
◊ подставляемая 101
◊ полиморфная 155, 391, 396
◊ получение текущего каталога 229
◊ посылки сообщения 438
◊ предикат 95, 214, 402
◊ преобразования 124, 180, 313
◊ просмотра 243
◊ прототип 63, 69
◊ расшифровки 251
◊ регистрации класса окна 426
◊ результат 115
◊ рекурсивная 116, 272, 273, 274, 278, 289
◊ с переменным числом параметров 264, 265, 395
◊ системная 354
◊ скалярное произведение 335
◊ сокращения 302
◊ сортировки 392
◊ спецификация исключений 353
◊ сравнения 218
◊ стандартная 62, 71, 354
◊ статическая 414
◊ суммирования 136, 143
◊ тело 67, 69

◊ универсальная 176
◊ фильтр 403
◊ чтения 252
◊ шаблон 92, 115
◊ элемент структуры 135
◊ эффективная 283

X

Хендл 420

Ц

Цикл обработки сообщений 425, 428, 443

Ч

Числа 34
◊ беззнаковые 40, 47, 58
◊ диапазон значений 35, 40, 41, 58
◊ длинные 40
◊ дробные 34, 37, 38, 40
◊ знаковые 40, 47
◊ короткие 40
◊ мантисса 35
◊ перевод 123
◊ размер 35, 40, 41, 58
◊ с плавающей точкой 34
◊ суффиксы 36
◊ тип 58
◊ точность 35
◊ целые 40, 58
◊ экспонента 35

Ш

Шаблон 345, 387
◊ вектора 197
◊ инстанцирование 93, 369, 413
◊ класса 369
◊ метапрограммирование 410
◊ параметры 133
◊ предикат 403
◊ специализация 412
◊ структуры 133, 172
◊ функции 92, 114, 144, 411
◊ функции-объекта 402
Ширина поля вывода 242
Шрифт системный 434

Э

Элемент управления 430, 434
◊ выпадающий список 444, 446
◊ кнопка 431, 445
◊ меню 431
◊ метка 450
◊ свойства 444
◊ список 430
◊ статический 435
◊ флажок 430, 444, 445
Эффективность рекурсивной функции 279