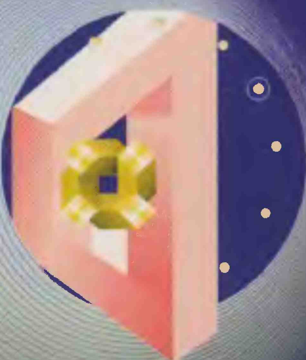
Г. Шилдт

Теория и практика C++

Секреты мастерства, позволяющие в полной мере использовать преимущества языка C++

- Методы параметризации
- Классы алгоритмов сортировки
- Шифрование и сжатие данных
- Стандартные классы строк
- Разбор математических выражений
- Разреженные массивы
- Шаблоны и их функции



МАСТЕР

РУКОВОДСТВО ДЛЯ ПРОФЕССИОНАЛОВ

Schildt's Expert C++

Herbert Schildt

УДК 681.3.06

Книга Герберта Шилдта, одного из самых известных авторов компьютерной литературы, посвящена обсуждению сложных вопросов программирования. В ней подробно рассмотрены параметризованные функции и классы, а также такие задачи программирования, как реализация разреженных массивов, построение программ разбора математических выражений, алгоритмы шифрования и сжатия данных, а также проблемы разработки собственных языков программирования и написания интеграторов для них. Приводятся ценные практические советы по приемам и методам работы, а также тексты готовых работающих программ.

Для профессиональных программистов

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Елизавета Кароник*
Перевод с английского и научное редактирование	*Ольга Кокорева*
Корректор	*Наталья Чухутина*
Дизайн обложки	*Игорь Шинкар*
Верстка	*Андрей Володов*

Г. Шилдт

Теория и практика С++: пер. с англ. — СПб.: BHV — Санкт-Петербург, 1996. — 416 с., ил.

Authorized translation from the English language edition published by Osborne McGraw-Hill Copyright © 1995. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission in writing from the Publisher. Russian language edition published by BHV — St. Petersburg. Copyright © 1996.

Авторизированный перевод английской редакции, выпущенной Osborne McGraw-Hill Copyright © 1995. Все права защищены. Никакая часть настоящей книги не может быть воспроизведена или передана в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на то нет письменного разрешения Издательства. Русская редакция выпущена BHV — Санкт-Петербург. Copyright © 1996.

ISBN 0-07-882209-2
ISBN 5-7791-0029-2

© 1995 by Osborne McGraw-Hill
© Перевод на русский язык "BHV — Санкт-Петербург", 1996

Лицензия ЛР № 090141 от 12.02.96. Подписано в печать 17.12.96.
Формат $70 \times 100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 33,6. Тираж 10 000 экз. Заказ № 279.
BHV — Санкт-Петербург, 195009, С.-Петербург, Бобруйская ул., 4.

Отпечатано с диапозитивов в ГПП «Печатный Двор»
Комитета РФ по печати.
197110, Санкт-Петербург, Чкаловский пр., 15.

Содержание

Введение **11**

Глава 1. Использование параметризованных функций **13**

Почему следует использовать параметризованные функции? 14
Методы параметризации без использования шаблонов 15
Изобретение шаблонов 17
Обзор функций шаблонов 18
Эффективность шаблонов функций 19
Построение параметризованных функций сортировки 20
Классы алгоритмов сортировки 21
Сравнительный анализ алгоритмов сортировки 21
Пузырьковая сортировка — злой дух перестановок 22
Сортировка методом отбора 26
Сортировка методом вставки 28
Усовершенствованные методы сортировки 30
Сортировка методом Шелла 31
Метод быстрой сортировки 33
Сравнение алгоритма быстрой сортировки со стандартной функцией qsort( ) 36
Сортировка типов, определенных пользователем 38
Выбор метода сортировки 41
Поиск 42
Методы поиска 42
Последовательный поиск 42
Бинарный поиск 43
Рекомендации для самостоятельной разработки 45

Глава 2. Исследование параметризованных классов **46**

Обзор параметризованных классов 47
Ограниченные массивы 48
Перегрузка оператора [ ] 48
Построение параметризованного ограниченного массива 49

Очереди 51
Циклическая очередь 54
Стеки 57
Связные списки 63
Построение параметризованного класса списка с двойными связями 64
Функция store( ) 67
Функция remove( ) 68
Отображение списка 69
Поиск объекта в списке 71
Полный листинг параметризованного класса связного списка с двойными ссылками 71
Бинарные деревья 79
Параметризованный класс дерева 81
Добавление элементов в дерево 82
Прохождение дерева 83
Поиск по дереву 87
Удаление элемента дерева 87
Рекомендации для самостоятельной разработки 88

Глава 3. Объектно-ориентированная программа разбора математических выражений 90

Выражения 91
Разбор выражений: постановка проблемы 92
Разбор выражения 93
Класс Parser 95
Разбиение выражений 96
Простая программа разбора выражений 98
Принципы работы программы разбора выражений. 104
Включение в программу разбора выражений возможность работы с переменными 105
Синтаксическая проверка в рекурсивно-нисходящем алгоритме разбора выражений 114
Построение параметризованной версии программы разбора выражений 115
Рекомендации для самостоятельной разработки 123

Глава 4. Разреженные массивы в стиле С++ 125

Цели разработки разреженных массивов 126
Объекты типа разреженных массивов 127
Разреженный массив на базе связного списка 128
Некоторые детали использования связного списка 133
Анализ подхода с использованием связного списка 135

Разреженным массивы на основе бинарных деревьев 136
Некоторые детали реализации разреженных массивов на основе бинарных деревьев 140
Анализ метода реализации разреженных массивов на основе бинарных деревьев 141
Разреженные массивы на основе массивов указателей 142
Некоторые детали реализации разреженных массивов с помощью массива указателей 145
Анализ метода, использующего массив указателей 145
Хэширование 146
Некоторые детали хэширования 153
Анализ хэширования 156
Выбор подхода к реализации разреженных массивов 157
Рекомендации для самостоятельной разработки 158

Глава 5. Принципы работы с информацией типа Run-time и ее использование 159

Зачем нужна информация RTTI? 160
Использование механизма typeid 166
Использование typeid для исправления программы обработки координат 168
Использование механизма dynamic_cast 171
Операторы преобразования типов. 171
Использование оператора dynamic_cast для исправления функции quadrant( ) 172
Применение RTTI 174
Рекомендации для самостоятельной разработки 181

Глава 6. Строки: использование стандартного класса строк .. 182

Почему стандартный класс string включен в определение C++? 183
Конструкторы строк 184
Операторы класса string 185
Некоторые функции-члены класса string 189
Присваивание и добавление частей строк 189
Вставка, удаление и замена 191
Поиск подстрок 193
Сравнение частей строк 195
Получение длины строки 196
Получение строки, завершающейся нулем 196
Простой строкоориентированный редактор, использующий класс string 198
Некоторые детали работы редактоа 207
Рекомендации для самостоятельной разработки 207

Глава 7. Шифрование и сжатие данных 208
Краткая история криптографии 209
Три основных типа шифров 210
Шифры замены 211
Алгоритмы перестановок 220
Шифры битовых манипуляций... 226
Сжатие данных 232
Преобразование 8-битного набора символов в 7-битный 233
Преобразование 4-байтового набора в 3-байтовый 237
Рекомендации для самостоятельной разработки 241

Глава 8. Интерфейс с функциями языка ассемблера 243
Для чего нужно использование языка ассемблера? 244
Основные принципы интерфейса с языком ассемблера 245
Соглашения о вызовах для компилятора С++ 247
Соглашения о вызовах для Microsoft/Borland С++ 247
Несколько слов о моделях памяти 249
Разработка функции на ассемблере 251
Передача аргументов функции 252
Вызов библиотечных функций и операторов 258
Получение доступа к структурам и классам 263
Использование указателей и ссылок 270
Пример, использующий гигантскую (huge) модель памяти 275
Ручная оптимизация 278
Построение основы для кода на ассемблере 280
Использование asm 284
Рекомендации для самостоятельной разработки 285

Глава 9. Создание и интеграция новых типов данных 286
Теория множеств 287
Операции над множествами 288
Определение типа множества 289
Конструкторы и деструктор класса Set 291
Добавление и удаление членов множества 292
Определение членства 293
Конструктор Copy 294
Присваивание для множеств 295
Перегрузка оператора + ... 296
Добавление нового элемента в состав множества 297
Построение объединения множеств 298
Перегрузка оператора - 299
Удаление элемента из множества 299
Разность множеств 300

Пересечение множеств 301
Симметричная разность 301
Определение равенства, неравенства и подмножества 302
Определение членства 303
Преобразование в целое 304
Перегрузка операторов ввода/вывода 304
Демонстрационная программа работы с множествами 305
Рекомендации для самостоятельной разработки 306

Глава 10. Реализация языковых интерпретаторов на C++ 317

Модуль разбора выражений Small BASIC 318
Выражения Small BASIC 319
Элементы Small BASIC 320
Программа разбора выражений Small BASIC 324
Как программа разбора выражений обрабатывает переменные 334
Интерпретатор Small BASIC 335
Ключевые слова 335
Загрузка программы 337
Главный цикл 338
Функция присваивания 339
Команда PRINT 340
Команда INPUT 342
Команда GOTO 343
Утверждение IF 346
Цикл FOR 347
Конструкция GOSUB 350
Полный код интерпретатора 352
Использование Small BASIC 363
Дополнение интерпретатора и расширение его возможностей 365
Разработка собственного языка программирования 366

Глава 11. От C++ к Java 367

Что представляет собой Java? 368
Почему Java? 368
Защищенность 369
Переносимость 369
Магическое решение Java: Java Bytecode 369
Различия между Java и C++ 370
Какие возможности отсутствуют в Java? 371
Что было добавлено в Java? 372
Пример программы на Java 373

Методы вместо функций 375
Компиляция программы на Java 375
Второй пример 375
Работа с классами 377
Финализаторы 379
Иерархия классов Java 380
Классы и файлы 383
Пакеты и импорт 383
Интерфейсы 384
Стандартные классы 385
Рекомендации для самостоятельной разработки 385

Приложение А. Список ключевых слов С ++ 386

Index 409

Введение

Сколь бы широкое распространение ни получили за последнее время CASE-средства высокого уровня и графические инструменты, подобные Visual Basic, но для написания программ всегда будут нужны программисты, а серьезные программы надо писать на C++. Если вам необходимы эффективные программы на C++, то для этого необходим самый свежий опыт объектно-ориентированного программирования. Книга Герберта Шилдта поможет вам овладеть секретами мастерства программирования и наряду с этим — даст вам гораздо больше. Книга не просто предлагает по-новому взглянуть на общеизвестные вещи и общепризнанные факты, не просто учит более эффективным приемам и методам работы — она побуждает к самостоятельным исследованиям и творчеству.

В конце каждой из глав этой книги приведены рекомендации автора для самостоятельной разработки. Используя приведенные в книге многочисленные (и реально работающие) примеры как начальную точку разработки, вы получаете широчайший простор для творчества. Разумеется, книга предназначена в первую очередь для опытных программистов, тех, кого называют "экспертами". Однако, даже если вы — новичок и только начали осваивать язык C++, не пугайтесь! Герберт Шилдт в высшей степени одарен удивительным талантом — доходчиво и просто говорить о сложных вещах. Вы с удивлением обнаружите, что, хотя в книге обсуждаются серьезные и сложные проблемы, в ней нет абсолютно ничего такого, что было бы недоступно для понимания. Наконец, она просто на редкость увлекательна и интересна. Именно поэтому книга по-своему уникальна — любой программист, каким бы ни был уровень его подготовки — от начинающего до эксперта, — найдет здесь для себя что-то интересное.

Глава 1

Использование параметризованных функций

Вероятно, самой важной из новых возможностей C++ является *шаблон* (**template**). Причина этого заключается в том, что шаблоны фундаментально изменяют внешнюю сторону программирования. Используя шаблон, вы можете создавать обобщенные спецификации для функций и для классов, которые часто называются *параметризованными функциями* (generic functions) и *параметризованными классами* (generic classes). Параметризованная функция определяет общую процедуру, которая может быть применена к данным различных типов. Параметризованный класс определяет общий класс, который может применяться к изменяющимся типам данных. В обоих случаях конкретный тип данных, над которыми выполняется операция, передается в качестве параметра. Как вы можете себе представить, именно это и делает шаблоны чрезвычайно ценным средством. В C++ шаблон функции или класса декларируется с помощью ключевого слова **template**. Возможно, вы уже знакомы с основными принципами использования ключевого слова **template**. В этой главе будут рассмотрены некоторые приемы, позволяющие в полной мере использовать преимущества, предоставляемые шаблонами.

Как уже говорилось, ключевое слово **template** используется для построения параметризованных функций и классов. В этой главе исследуются параметризованные функции, а параметризованные классы будут изучаться в следующей главе. Мы исследуем причины, вследствие которых были изобретены шаблоны, выясним, почему использование шаблонов представляет собой более эффективный метод по сравнению с другими средствами создания параметризованных функций, а затем на ряде примеров проиллюстрируем их применение.

Для демонстрации возможностей шаблонов мы используем *поиск* и *сортировку* - две наиболее широко распространенных в компьютерном мире операции. Алгоритмы поиска и сортировки выбраны нами по трем причинам. Во-первых, они легко преобразуются в форму шаблонов, и, как вы увидите далее, это преобразование существенно повышает их эффективность. Во-вторых, поиск и сортировка используются практически в любой программе, и поэтому чтение этой главы принесет практическую пользу всем программистам, которые в результате этого получат в свое распоряжение разнообразные параметризованные функции поиска и сортировки. Наконец, алгоритмы сортировки представляют собой одну из самых интересных и захватывающих областей науки программирования. Кроме того, эту тему многие не разрабатывают, а принимают на веру. Так, многие программисты используют в своих разработках стандартные библиотечные функции сортировки. Однако, как вы вскоре убедитесь, благодаря использованию шаблонов, на эту важную тему стоит посмотреть свежим взглядом.

Почему следует использовать параметризованные функции?

С первых же дней существования программирования программисты понимали необходимость в параметризованных подпрограммах, которые можно использовать повторно. Как вы знаете, многие алгоритмы не зависят от типа данных, которыми они манипулируют. Например, рассмотрим следующую последовательность обмена значениями между двумя переменными:

```
DataType temp, x, y;
// ...
temp=x;
x=y;
y=temp;
```

Если не рассматривать случаи, заведомо содержащие внутреннюю ошибку, этот алгоритм работает вне зависимости от фактического значения типа данных DataType. К примеру, алгоритм работает одинаково как при обмене целыми значениями, так и при обмене значениями типа **double** или **long**. Таким образом, логика алгоритма одинакова для всех типов данных. Однако, в большинстве языков программирования для обмена данными каждого типа требуется написание новой версии подпрограммы, несмотря на то, что лежащий в ее основе алгоритм остается неизменным. Эту ситуацию можно обобщить. Многие алгоритмы допускают отделение метода от данных. При использовании таких алгоритмов большим преимуществом была бы возможность однократного определения и отладки логики алгоритма, и последующее применение алгоритма к различным типам данных без необходимости перепрограммирования. Это не толь-

ко позволяет экономить усилия и время, но и страхует от ошибок, так как во многих случаях к разнообразным ситуациям будет применяться один и тот же отлаженный код.

Методы параметризации без использования шаблонов

Обобщенные параметризованные процедуры предоставляют очевидные преимущества. Неудивительно поэтому, что программисты всегда пытались их использовать. Однако, до изобретения шаблонов, такие попытки имели только частичный успех. Причину этого понять легко: в распоряжении программистов не было хорошего способа построения параметризованной функции. Поскольку программисты, как правило, изобретательны, а желание использовать параметризованные функции всегда было достаточно велико, появилось два далеких от совершенства метода. Первый из них заключался в построении параметризованных функций через использование функционально-подобного макроса. Например, нижеприведенный макрос создает "родовую функцию", выполняющую преобразование в отрицательное число:

```
#define neg(a)   (((a)<0) ? -(a) : (a))
```

Поскольку компилятор просто выполняет текстовую замену, когда встречается этот (или любой другой) макрос, функционально-подобный макрос **neg()** может использоваться с любым из встроенных типов данных. Например, допустимыми являются оба из приведенных ниже вызова макроса **neg()**:

```
char x;
float f;

x=neg(-10);
f=neg(123.23);
```

Однако, добиться работы этого макроса с типами данных, определенными пользователем, достаточно сложно. Кроме того, поскольку этот макрос не выполняет никакой проверки типов, возможны ситуации, когда он будет ошибочно использован с такими типами данных, для которых операция преобразования к отрицательному числу не определена (например, со строками). Поэтому, хотя функционально-подобные макросы удобны для небольших функций, в целом они не являются удачным решением.

Второй метод построения параметризованной функции заключался в добавлении одного или нескольких параметров, предназначенных для определения типов данных, над которыми функция выполняет операции. Например, распространенным подходом были передача функции указателя на данные в качестве одного параметра, и размера этих данных в байтах — в качестве другого параметра. Внутри функции параметр, определяющий размер данных, использовался

для предоставления функции возможностей обработки различных типов данных. Разумеется, использование дополнительного параметра означает генерацию избыточного кода при вызове функции. Параметры функции передаются через стек. Передача каждого параметра генерирует несколько инструкций в машинном коде. Таким образом, каждый дополнительный параметр уменьшает эффективность кода, увеличивая время, требующееся на вызов функции. Если функция вызывается постоянно, это снижение эффективности быстро создаст проблему. Поэтому ранее параметризация функции автоматически означала замедление ее работы. Таким образом, за преимущества, предоставляемые параметризацией функции, приходилось дорого платить.

Для того, чтобы проиллюстрировать дальнейшее обсуждение конкретным примером, рассмотрим хорошо известный образец функции, параметризованной в старом стиле — функцию **qsort()**. Как вы, возможно, уже знаете, **qsort()** является стандартной библиотечной функцией сортировки языка C++. Она выполняет сортировку данных любых типов, используя алгоритм быстрой сортировки. (Далее в данной главе вы узнаете, как реализовать собственную версию алгоритма быстрой сортировки.) Однако, поскольку функция **qsort()** заимствована из библиотеки стандартного C, она использует устаревший подход к параметризации, основанный на передаче параметров. Прототип функции приведен ниже:

```
void qsort(void *buf, size_t num, size_t size,
           int (*comp)(const void*, const void*));
```

Здесь параметр *buf* является указателем на массив, содержащий данные, которые должны быть отсортированы. Количество сортируемых элементов передается параметром *num*. Для поддержки алгоритма сортировки технически необходимыми являются только два параметра. Однако, в целях параметризации функции ей передается параметр *size*, определяющий размер элемента массива, и параметр *comp*, представляющий собой указатель на функцию, осуществляющую сравнение двух элементов. Таким образом, параметризация функции требует добавления двух параметров. Если бы функция использовалась эпизодически, это не являлось бы предметом, достойным обсуждения. Однако, как будет видно из дальнейшего изложения, функция **qsort()** обычно реализуется как *рекурсивная* функция, и поэтому наличие дополнительных параметров приводит к значительному снижению ее производительности.

В течение многих лет преимущества, предоставляемые использованием параметризованных функций, практически сводились на нет результирующим снижением эффективности, что препятствовало широкому использованию таких функций в масштабных разработках. Однако, ввод шаблонов изменил ситуацию к лучшему.

Изобретение шаблонов

Хотя функции шаблонов и ключевое слово **template** представляют собой одну из важнейших особенностей языка, изначально они не являлись частью языка C++. Фактически спецификация языка C++ в течение нескольких лет даже не включала в свой состав ключевого слова **template**. Тем не менее, необходимость разработки некоего метода определения параметризованных функций была общепризнанной. Ядром параметризованных функций и классов является концепция *параметризованного типа* (parameterized type). Проблема заключалась только в том, как наилучшим образом реализовать эту концепцию. Б. Страуструп в своей книге "*The Design and Evolution of C++*" (Addison-Wesley, 1994) пишет: "В исходном проекте языка C++ параметризованные типы учитывались, однако, их реализация была отложена из-за нехватки времени на тщательное исследование и полноценную реализацию". Тем не менее, параметризованные типы (позволяющие создавать параметризованные функции и классы) всегда считались исключительно важными для дальнейшего долгосрочного развития и распространения C++. Параметризованные типы особенно важны для поддержки параметризованных *контейнерных классов* (container classes), которые подробно обсуждаются в главе 2. Двигаясь в этом направлении, Страуструп начал определять шаблоны в 1986 году. К 1990 году он представил экспериментальную разработку, реализующую эту возможность, в своей книге *Annotated C++ Reference Manual* (Addison-Wesley, 1990). Именно эта версия определения шаблона (**template**) *в* 1990 году была добавлена к развивающемуся стандарту ANSI C++. Теперь эта возможность поддерживается всеми основными компиляторами C++.

Разработка шаблонов преследовала цель добиться выигрыша сразу по трем пунктам: эффективности, защищенности типов и простоте использования. Страуструп хотел сделать функции шаблонов такими же эффективными, как и их непараметризованные версии. Как уже упоминалось, неэффективные средства параметризации функций к этому времени уже существовали, и требовалось только реализовать эффективный подход. Кроме того, Страуструп хотел сохранить проверку типов аргументов и параметров функции. Как уже упоминалось, отсутствие проверки типов представляло собой фундаментальную проблему при использовании функционально-подобных макросов. Наконец, Страуструп считал, что использование шаблонов должно быть простым с точки зрения программиста. Возможно, ваш собственный опыт программирования также говорит вам о том, что если языковая возможность сложна в использовании, то пользоваться ею будут немногие. Таким образом, простота и удобство использования играют ключевую роль при добавлении в язык программирования любого нового элемента. К счастью, мастерство Страуструпа позволило ему осуществить все эти цели. Реализация шаблонов представляет собой масштабное достижение, представляющее собой существенное дополнение к языку, как в смысле функциональности, так и в эстетическом совершенстве.

Обзор функций шаблонов

Прежде, чем начинать применять шаблоны, необходимо хотя бы обзорно ознакомиться с их синтаксисом и возможностями. Параметризованная функция создается с помощью ключевого слова **template**. Язык С++ в точности отражает обыденный смысл этого слова. Слово template используется для создания шаблона (или каркаса), описывающего в общих чертах назначение функции и предоставляющего компилятору указывать конкретные подробности по мере необходимости.

Шаблон определяет общий набор операций, которые будут применяться к данным различных типов. При этом тип данных, над которыми функция должна выполнять операции, передается ей в виде параметра на *стадии компиляции*. Благодаря использованию этого механизма одна и та же общая процедура может применяться к широкому диапазону типов данных. Создавая шаблон, вы определяете — вне зависимости от данных — сущность алгоритма. После того, как это будет сделано, компилятор будет автоматически генерировать правильный код, соответствующий типу данных, который фактически используется при вызове функции. В частности, когда вы создаете родовую функцию, вы создаете функцию, которая может автоматически перегружать сама себя.

Общая форма функции-шаблона **template** приведена ниже:

template <class *Ttype*> *ret-type func-name*(*parameter list*)

{

// body of function

}

Здесь параметр Ttype обозначает тип данных, используемых функцией. Это имя может использоваться в пределах действия определения функции. Когда компилятор будет создавать конкретную версию этой функции, он автоматически заменит этот параметр конкретным типом данных. Утверждение **template** позволяет определить несколько родовых типов данных, которые в списке должны отделяться друг от друга запятыми.

Ниже приведен пример параметризованной функции, которая осуществляет обмен значениями между двумя ее параметрами:

```
template <class SwapType> void swap(SwapType &x, SwapType &y)
{
    SwapType temp;

    temp=x;
    x=y;
    y=temp;
}
```

Эту функцию можно вызывать с переменными любых типов. Компилятор автоматически создаст соответствующий конкретный экземпляр функции.

При обсуждении шаблонов используются и другие термины, которые вы, скорее всего, встретите в литературе по C++. Конкретную версию функции, создаваемую компилятором, называют порожденной функцией (*generated function*). Процесс построения порожденной функции называется конкретизацией (*instantiating*). Иными словами, порожденная функция представляет собой конкретный экземпляр, принадлежащий к семейству родственных функций, задаваемых шаблоном.

Хотя функция-шаблон по мере надобности может перегружать себя сама, вы можете выполнять и ее явную перегрузку. Если вы перегружаете параметризованную функцию, то эта перегруженная функция "скрывает" параметризованную функцию по отношению к конкретной версии. Ручная перегрузка шаблона позволяет вам настроить версию параметризованной функции таким образом, чтобы она соответствовала конкретной ситуации. Однако, в общем случае, если вам необходимо иметь для различных типов данных различные версии функции, лучше использовать не шаблоны, а перегруженные функции.

Эффективность шаблонов функций

В отличие от устаревших методов построения параметризованных функций шаблоны чрезвычайно эффективны. До включения в спецификацию языка C++ ключевого слова **template** единственным способом построения параметризованной функции была передача функции информации о типе данных через параметры (как описывалось ранее). В пределах функции эта избыточная информация использовалась для того, чтобы предоставить функции возможность манипулировать различными типами данных. Как уже разъяснялось ранее, этот подход изначально неэффективен из-за дополнительных параметров и увеличения объема вычислений, которые должны быть проделаны в функции. Шаблоны позволяют снять эту проблему. За счет чего это достигается?

Каждый раз, когда вы конкретизируете шаблон, строя порожденную функцию, компилятор автоматически создает нужную версию этой функции. Это означает, что компилятор автоматически создает перегруженную версию параметризованной функции, в которой параметризованные типы данных заменены реальными типами, над которыми должны осуществляться действия. Это означает, что с порожденной функцией не ассоциируются дополнительные накладные расходы (дополнительные параметры, избыточный код). Порожденная функция будет настолько же эффективна, как если бы вы написали ее версию для данного конкретного случая. Вся разница заключается только в том, что теперь этот процесс автоматизирован. Как вы увидите далее, разница в эффективности, обеспечиваемой параметризованными функциями в старом стиле и шаблонами, оказывается весьма существенной.

Построение параметризованных функций сортировки

Сортировка (sorting) - это процесс, позволяющий упорядочить множество подобных данных в возрастающем или убывающем порядке. Сортировка представляет собой один из наиболее приятных с интеллектуальной точки зрения алгоритмов, поскольку процесс хорошо определен. Алгоритмы сортировки хорошо исследованы и изучены. К сожалению, по этой причине сортировку иногда принимают как некую данность. Фактически, когда возникает необходимость в сортировке данных, большинство программистов просто используют стандартную функцию сортировки, имеющуюся в библиотеке, которая поставляется с их компилятором, и больше об этом не задумываются. Однако, как вы увидите далее, различные подходы к сортировке обладают различными характеристиками. Хотя некоторые методы в среднем могут быть лучше других, ни один из методов не будет идеальным для всех ситуаций. Поэтому каждый программист должен иметь в своем распоряжении несколько различных типов сортировки. То, что теперь эти методы могут быть реализованы в виде параметризованных функций, еще более усиливает их значимость.

Большинство компиляторов С++ поддерживают стандартную функцию **qsort()**, которая является частью их стандартной библиотеки. Хотя эта функция, как правило, реализована достаточно эффективно, но после прочтения данной главы вы не захотите ее использовать по двум причинам. Во-первых, функция **qsort()** унаследована из библиотеки С. Это значит, что хотя она и может сортировать большинство типов данных, это достигается довольно громоздкими методами и вручную. Используя параметризованную функцию сортировки, вы можете создать параметризованный шаблон, который будет использоваться для построения конкретных версий сортировки для каждого типа данных. При этом исключаются избыточные вычисления, свойственные подходу, реализованному стандартной функцией **qsort()**. Во-вторых, хотя алгоритм быстрой сортировки (реализованный функцией **qsort()**) в общем случае довольно эффективен, он не является наилучшим для некоторых ситуаций. Поэтому, для того чтобы обеспечить для всех ситуаций наивысшую скорость сортировки, вам необходимо знать несколько алгоритмов сортировки.

Существуют две основные категории алгоритмов сортировки: алгоритмы, которые сортируют массивы, и алгоритмы, которые сортируют последовательные файлы на диске или ленте. Поскольку среднему программисту наиболее интересна первая из этих категорий, именно она и будет рассматриваться в данной главе.

Наиболее часто при сортировке информации в качестве ключа сортировки используется только часть этой информации. Ключ сортировки представляет собой часть данных, определяющую место элемента данных в общем списке. Таким образом, при сравнениях используется только ключ сортировки, однако при упорядочении списка данных они переносятся на новую позицию целыми структу-

рами. Например, в почтовом списке рассылки в качестве ключа сортировки можно использовать только поле ZIP-кода, однако в процессе сортировки будет изменяться порядок перечисления полных адресов. Последующие несколько примеров, в целях упрощения изложения, сортируются только массивы, состоящие из встроенных типов, в которых ключ сортировки и данные совпадают. Далее мы рассмотрим применение этих методов для сортировки любых определенных пользователем типов.

Классы алгоритмов сортировки

Хотя существует множество методов сортировки данных, все эти методы можно отнести к одному из следующих классов:

- Перестановка
- Отбор
- Вставка

Принципы, лежащие в основе этих методов, проще всего уяснить себе на примере колоды карт. Для того, чтобы выполнить сортировку колоды карт методом перестановки, разложите карты на столе, а затем обменивайте их местами до тех пор, пока колода не будет разложена в нужном порядке. Для иллюстрации метода отбора, разложите карты на столе, выберите карту наименьшей значимости, извлеките ее из общей колоды и держите в руке. Затем из карт, оставшихся на столе, выберите следующую наименее значимую, и поместите ее за картой, которая уже находится у вас в руке. Продолжайте этот процесс до тех пор, пока в руке у вас не окажутся все карты. Колода, которая будет находиться у вас в руке после окончания этого процесса, будет отсортирована. Сортировку методом вставки иллюстрирует следующий пример. Держа колоду в руке, извлекайте из нее по одной карте, при этом всегда помещайте извлекаемую из колоды карту в нужную позицию. По окончании этого процесса, когда в руке у вас не окажется карт, колода будет отсортирована.

Для того, чтобы лучше понять разницу между всеми этими подходами, мы разработаем алгоритмы сортировки для каждой из перечисленных категорий.

Сравнительный анализ алгоритмов сортировки

Как можно оценить относительные преимущества одного метода сортировки над другим при наличии такого их разнообразия? Разумеется, скорость исполнения сортировки важна, но многие методы сортировки обладают уникальными характеристиками, влияющими на их применимость в том или ином случае. Таким образом, иногда среднюю скорость выполнения сортировки приходится сопоставлять с другими факторами. Ниже приведены четыре критерия, которые следует использовать при выборе метода сортировки.

- Какова его средняя скорость сортировки?
- Какова его скорость в наилучшем и наихудшем случаях?
- Естественно или нет его поведение?
- Выполняет ли он сортировку элементов с совпадающими ключами?

Рассмотрим эти критерии более подробно. Очевидно, что скорость конкретного алгоритма сортировки имеет важное значение. Скорость сортировки массива непосредственно зависит от количества выполняемых сравнений и следующих за ними перестановок. *Сравнение* имеет место в случае, когда два элемента массива сравниваются между собой; *перестановкой* же называется операция, при которой два элемента массива меняются местами. Как будет видно далее, для некоторых методов сортировки время исполнения возрастает в экспоненциальной зависимости от количества сортируемых элементов, в то время как для других эта зависимость будет логарифмической.

Наилучшая и наихудшая скорость важны в том случае, если вы имеете основания ожидать частого повторения одной из этих ситуаций. Очень часто при хорошей средней скорости алгоритмы обладают неприемлемой наихудшей.

Под *естественностью поведения* понимается следующее: если список уже упорядочен, алгоритм должен выполнять минимальное количество операций, если список не упорядочен, алгоритм должен выполнять больший объем работы, и наконец, наибольшую работу он должен проделывать, если список отсортирован в обратном порядке. Интенсивность работы алгоритма основана на количестве сравнений и перестановок, которые он выполняет.

Для того, чтобы понять, почему может иметь значение перестановка элементов с одинаковыми ключами, представьте себе базу данных (например, почтовый список рассылки), которая сортируется по главному ключу и подключу. Основным ключом является ZIP-код, а в пределах ZIP-кода подключом является фамилия. При добавлении в базу нового адреса и следующей за этим пересортировке списка вы не хотите, чтобы выполнялась пересортировка по подключам. Для того, чтобы гарантировать, что этого не произойдет, алгоритм не должен выполнять перестановку главных ключей с одинаковыми значениями.

Пузырьковая сортировка — злой дух перестановок

Самым известным (и самым скверным) методом сортировки является *пузырьковый метод*. Его популярность является следствием запоминающегося названия и простоты. Однако, в общем случае это — самый худший из всех когда-либо изобретенных методов.

Пузырьковая сортировка основана на методе перестановок. Ее смысл заключается в постоянном сравнении смежных элементов и при необходимости — их перестановке. Элементы массива в этом методе уподобляются всплывающим пузырькам в резервуаре с водой — каждый из них достигает своего уровня. Для начала реализуем непараметризованную форму пузырьковой сортировки. Таким образом вы сможете лучше понять принципы работы этого алгоритма. Нижеприведенная программа содержит непараметризованную версию пузырьковой сортировки, которая может использоваться для сортировки символьных массивов.

```
//Простейшая реализация пузырьковой сортировки для символов
#include <iostream.h>
#include <string.h>
void bubble(char *item, int count);
main()
{
    char str[] = "dcab";
    bubble(str, (int) strlen(str));
    cout << "Отсортированная строка: " << str << endl;
    return 0;
}
//Простая версия пузырьковой сортировки
void bubble(char *item, int count)
{
    register int a, b;
    char t;
    for(a=1; a<count; ++a)
        for(b=count-1; b>=a; - -b) {
            if(item[b-1] > item[b]) {
                //Перестановка значений
                t=item[b-1];
                item[b-1] = item[b];
                item[b] = t;
                }
            }
}
```

В функции **bubble( )**, **item** представляет собой указатель на массив символов, который должен подвергнуться сортировке, а параметр **count** представляет собой количество элементов массива. Метод пузырьковой сортировки управляется двумя вложенными циклами. При условии, что массив содержит **count** элементов, внешний цикл вызывает сканирование массива **count**−1 раз. Это гарантирует, что даже в наихудшем случае после завершения функции каждый элемент будет находиться на своем месте. Внутренний цикл фактически выполняет сравнения и перестановки. (Несколько усовершенствованная версия пузырьковой сортировки завершается, если перестановок не происходит, но данная версия

выполняет сравнение на каждом проходе через внутренний цикл.) Эту версию пузырьковой сортировки можно использовать для сортировки массива символов в восходящем порядке.

Для того, чтобы лучше уяснить себе, как работает алгоритм пузырьковой сортировки, рассмотрим каждый проход через цикл:

Исходный массив	d c a b
проход 1	a d c b
проход 2	a b d c
проход 3	a b c d

Анализируя алгоритмы сортировки, вы должны определить, сколько сравнений и перестановок будет выполнено в лучшем, в среднем и в наихудшем случае. Для алгоритма пузырьковой сортировки количество выполняемых сравнений всегда одинаково, так как оба цикла **for** повторяются указанное количество раз вне зависимости от того, отсортирован список или нет. Это означает, что при пузырьковой сортировке всегда выполняется

$1/2(n^2-n)$

сравнений, где n — количество сортируемых элементов. Эта формула следует из того факта, что внешний цикл выполняется *n*-1 раз, а внутренний — *n*/2 раз. Перемножив эти значения, получаем вышеприведенную формулу.

В наилучшем случае (если список уже отсортирован) количество перестановок равно нулю. Количество перестановок в среднем и в худшем случаях равны соответственно:

В среднем случае $3/4(n^2-n)$

В худшем случае $3/2(n^2-n)$

Объяснение того, как получены эти формулы, выходит за рамки этой книги, однако, вы можете интуитивно предположить, что по мере того, как снижается степень упорядоченности списка, количество неупорядоченных элементов, нуждающихся в перестановке, приближается к числу выполняемых сравнений.

Пузырьковая сортировка называется *n-квадратичным алгоритмом*, так как время его выполнения пропорционально квадрату количества элементов сортируемого массива. Алгоритм чрезвычайно неэффективен при работе с большими массивами, так как время выполнения непосредственно связано с количеством выполняемых сравнений и перестановок. Пренебрежем временем, занимаемым на перестановку элемента, позиция которого не соответствует его значению, и допустим, что одна операция сравнения занимает 0.001 секунды. Тогда сортировка десяти элементов займет примерно 0.05 секунды, для 100 элементов потребуется уже около 5 секунд, а для 1000 элементов это время возрастет уже до 500 секунд. Сортировка 100000 элементов (небольшой телефонный справоч

ник) займет 5000000 секунд или около 1400 часов - только представьте себе — целых два месяца непрерывной сортировки! На рис. 1.1 показана зависимость времени выполнения от размеров сортируемого массива.

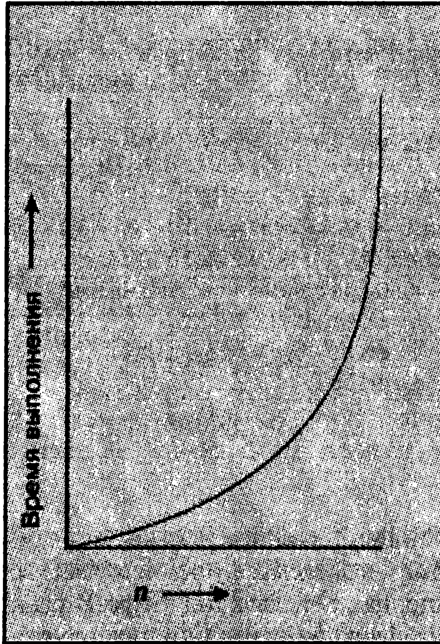


Рис. 1.1. Время выполнения пузырьковой сортировки находится в квадратичной зависимости от размера сортируемого массива

Преобразовать непараметризованную версию пузырьковой сортировки в шаблон достаточно просто. Для этого необходимо параметризовать тип сортируемых данных. Ниже приведена параметризованная версия пузырьковой сортировки:

```
//Параметризованная реализация пузырьковой сортировки
#include <iostream.h>
#include <string.h>
template <class Stype> void bubble(Stype *item, int count);
main()
{
    //сортировка массива символов
    char str[] = "dcab";
    bubble(str, (int) strlen(str));
    cout << "Отсортированная строка: " << str << endl;
    //сортировка массива целых чисел
    int nums[] = {5, 7, 3, 9, 5, 1, 8};
```

```
    int i;
    bubble(nums, 7);
    cout << "Отсортированный массив:";
    for(i=0; i<7; i++) cout << nums[i] << " ";
    return 0;
}
//Параметризованная версия пузырьковой сортировки
template <class Stype> void bubble(Stype *item, int count)
{
    register int a, b;
    Stype t;
    for(a=1; a<count; ++a)
        for(b=count-1; b>=a; - -b) {
            if(item[b-1] > item[b]) {
                //Перестановка значений
                t=item[b-1];
                item[b-1] = item[b];
                item[b] = t;
                }
            }
}
```

Как видите, теперь тип сортируемых данных указывается с помощью параметризованного типа **Stype**. Кроме того, в пределах функции **bubble()** временная переменная **t** также декларируется как принадлежащая к типу **Stype**. В функции **main()** при каждом вызове параметризованной функции **bubble()** компилятором генерируется ее экземпляр, соответствующий типу сортируемых данных. Так, в первом случае генерируется сортировка массива символов. Во втором случае порождается функция сортировки массива целых. Важно понимать, что в каждом конкретном случае компилятор замещает шаблон ***Stype*** фактически используемым типом данных. Таким образом, для каждого типа данных порождается своя функция сортировки. Именно поэтому параметризованную функцию алгоритма пузырьковой сортировки можно применять к данным любого типа. Прежде, чем продолжать, попробуйте поэкспериментировать и отсортируйте, скажем, массив чисел типа **double**.

Сортировка методом отбора

При сортировке методом отбора алгоритм находит элемент с наименьшим значением и выполняет перестановку, меняя его местами с первым элементом массива. После этого из оставшихся *n*-1 элементов ищется наименьший, после чего осуществляется его перестановка со вторым элементов и так далее. Перестановки будут осуществляться до тех пор, пока не поменяются местами последние два элемента. Например, если бы строка "dcab" сортировалась методом отбора, то каждый из проходов давал бы следующий результат:

Исходный массив	d c a b
проход 1	a c d b
проход 2	a b d c
проход 3	a b c d

Программа в нижеприведенном примере содержит параметризованную версию алгоритма сортировки методом отбора:

```
//Параметризованная реализация сортировки методом отбора
#include <iostream.h>
#include <string.h>
template <class Stype> void select(Stype *item, int count);
main()
{
     //сортировка массива символов
     char str[] = "dcab";
     select(str, (int) strlen(str));
     cout << "Отсортированная строка: " << str << endl;
     //сортировка массива целых чисел
     int nums[] = {5, 7, 3, 9, 5, 1, 8};
     int i;
     select(nums, 7);
     cout << "Отсортированный массив:";
     for(i=0; i<7; i++) cout << nums[i] << " ";
     return 0;
}
//Параметризованная функция сортировки отбором
template <class Stype> void select(Stype *item, int count);
{
     register int a, b, c;
     int exchange;
     Stype t;
     for(a=0; a<count-1; ++a) {
          exchange = 0;
          c=a;
          t = item[a];
          for(b=a+1; b<count; ++b) {
               if(item[b]<t) {
                    c = b;
                    t = item[b];
                    exchange = 1;
               }
          }
          if(exchange) {
               item{c] = item[a];
               item[a] = t;
          }
     }
}
```

К сожалению, сортировка методом отбора тоже является *n*-квадратичным алгоритмом. Внешний цикл исполняется *n*-1 раз, а внутренний — *n*/2 раз. В результате сортировка методом отбора требует $1/2(n^2-n)$ сравнений, что сильно замедляет работу метода при большом количестве элементов. Количества перестановок, требующееся в наилучшем и наихудшем случаях для метода сортировки отбором, будут следующими:

Наилучший случай: $3(n-1)$

Наихудший случай: $n^2/4+3(n-1)$

В наилучшем случае, если список уже упорядочен, требуется сравнить только (*n*-1) элемент, и каждое сравнение требует трех промежуточных шагов. Наихудший случай аппроксимирует количество сравнений. Средний случай для определения труден, и вывод его формулы выходит за рамки данной книги. Однако, формулу для его вычисления мы приведем:

$$n(\log n + y)$$

где *y* — константа Эйлера, примерно равная 0.577216.

Хотя количество сравнений для пузырьковой сортировки и сортировки методом отбора одинаковы, однако, для сортировки отбором показатель количества перестановок в среднем случае намного лучше. Тем не менее, существуют еще более совершенные методы сортировки.

Сортировка методом вставки

Метод вставки представляет собой третий и последний из основных алгоритмов сортировки. На первом шаге выполняется сортировка первых двух элементов массива. Далее алгоритм ставит третий элемент в порядковую позицию, соответствующую его положению относительно первых двух элементов. Затем в этот список вставляется четвертый элемент и т. д. Процесс продолжается до тех пор, пока все элементы не будут отсортированы. Например, если требуется отсортировать массив "dcab", процесс сортировки методом вставки будет состоять из следующих шагов:

Исходный массив	d c a b
Проход 1	c d a b
Проход 2	a c d b
Проход 3	a b c d

Нижеприведенная программа реализует параметризованную версию алгоритма сортировки вставкой.

```
//Параметризованная реализация сортировки методом вставки
#include <iostream.h>
#include <string.h>
```

```
template <class Stype> void insert(Stype *item, int count);
main()
{
     //сортировка массива символов
     char str[] = "dcab";
     insert(str, (int) strlen(str));
     cout << "Отсортированная строка: " << str << endl;
     //сортировка массива целых чисел
     int nums[] = {5, 7, 3, 9, 5, 1, 8};
     int i;
     insert(nums, 7);
     cout << "Отсортированный массив:";
     for(i=0; i<7; i++) cout << nums[i] << " ";
     return 0;
}
//Параметризованная функция сортировки вставкой
template <class Stype> void insert(Stype *item, int count);
{
     register int a, b;
     Stype t;
     for(a=1; a<count; ++a) {
          t = item[a];
          for(b=a-1; b>=0 && t<item[b]; b- -)
               item[b+1]=item[b];
          item[b+1]=t
     }
}
```

В отличие от пузырьковой сортировки и сортировки методом отбора количество сравнений, имеющих место в процессе сортировки методом вставки, зависит от исходной упорядоченности списка. Если список отсортирован в нужном порядке, то количество сравнений равно n-1. Если список не упорядочен, то количество сравнений равно

$1/2(n^2+n)$

В наилучшем, среднем и наихудшем случаях количество сравнений будет равно:

В наилучшем случае $2(n-1)$

В среднем $1/4(n^2+n)$

В наихудшем случае $1/2(n^2+n)$

По этой причине в наихудших случаях алгоритм вставки так же плох, как и пузырьковый метод или метод отбора; в среднем случае он лишь немногим лучше этих методов. Однако, алгоритм сортировки методом вставки обладает двумя реальными преимуществами. Во-первых, его поведение естественно. Это означает, что если массив уже отсортирован в нужном порядке, алгоритм проводит наименьшее количество вычислений, а если массив отсортирован в порядке,

обратном требуемому (наихудший случай), — его работа наиболее интенсивна. Благодаря этому алгоритм вставки отлично работает с почти упорядоченными массивами. Другим преимуществом является то, что этот метод не изменяет порядка следования одинаковых ключей. Это означает, что если список уже отсортирован по двум ключам, то после сортировки методом вставки он останется отсортированным по обоим ключам.

Несмотря на то, что для некоторых наборов данных количество сравнений может быть сравнительно невелико, при каждой вставке элемента на его место должен выполняться сдвиг массива. В результате количество перестановок может оказаться довольно значительным. Тем не менее, существуют и еще лучшие методы сортировки.

Усовершенствованные методы сортировки

Все алгоритмы, описанные в предыдущем разделе, имели один общий недостаток — время выполнения каждого из них находилось в квадратичной зависимости от количества элементов массива. При сортировке больших объемов данных это сильно замедляет работу алгоритма. Фактически при достижении некоторой критической точки все эти алгоритмы становятся недопустимо медленными. К сожалению, "страшные истории" о сортировке, которая "занимает три дня", имеют под собой почву. Если процесс сортировки чрезмерно затягивается, обычно в этом "виноват" алгоритм сортировки, применяемый в программе. Первой реакцией программиста на такое безобразие обычно бывает попытка ручной оптимизации кода с использованием языка ассемблера. Ручная оптимизация действительно может ускорить выполнение процедуры (обычно в постоянное число раз). Несмотря на это, если используемый алгоритм сортировки неэффективен, процесс все равно будет медленным, как бы оптимально ни было выполнено кодирование. Следует иметь в виду, что если скорость выполнения алгоритма пропорциональна n^2, увеличение тактовой частоты компьютера или скорости выполнения кода дает лишь незначительное улучшение, поскольку возрастание времени выполнения имеет экспоненциальный характер. (Фактически изображенная на рис. 1.1 зависимость n^2 немного сдвигается вправо, оставаясь по остальным показателям неизменной.) Следует понять, что если скорость выполнения процедуры возрастает экспоненциально, то не поможет никакая ручная оптимизация. Единственным выходом будет использование более эффективного алгоритма.

Следующие два раздела данной главы описывают два отличных метода сортировки — метод Шелла (Shell sort) и метод быстрой сортировки (quicksort), который считается наилучшим из существующих методов сортировки. Оба этих метода по своим показателям существенно превышают описанные ранее базовые алгоритмы.

Сортировка методом Шелла

Этот метод назван по имени его изобретателя (D.L. Shell). Он построен на основе метода вставки с минимизацией промежуточных шагов. Рассмотрим иллюстрацию на рис. 1.2. Сначала выполняется сортировка элементов, отстоящих друг от друга на три позиции. После этого сортируются элементы, отстоящие друг от друга на две позиции. Наконец, выполняется сортировка смежных элементов.

Первый взгляд на этот алгоритм не дает понимания того, почему этот кажущийся столь сложным метод дает столь хорошие результаты. Непонятно даже, как он вообще сортирует массив. Однако, метод работает. Каждый промежуточный шаг задействует относительно небольшое количество элементов, которые к тому же могут уже находиться в нужном порядке. Поэтому метод Шелла эффективен, и упорядоченность массива возрастает после каждого прохода.

Точная последовательность изменения приращений может изменяться. Единственным требованием остается равенство последнего приращения 1. Например, хорошо себя зарекомендовала последовательность 9, 5, 3, 2, 1, которая использована и в нижеприведенном примере реализации алгоритма Шелла. Избегайте

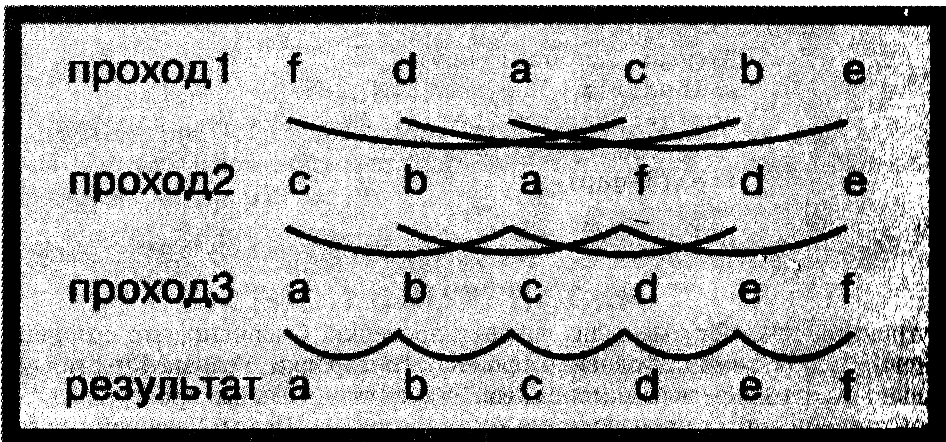


Рис. 1.2. Сортировка методом Шелла

последовательностей степеней 2, поскольку математически строго доказано, что это снижает эффективность алгоритма (который, тем не менее, работает и в этом случае).

```
//Параметризованная реализация алгоритма Шелла
#include <iostream.h>
#include <string.h>

template <class Stype> void shell(Stype *item, int count);
```

```
main()
{
     //сортировка массива символов
     char str[] = "dcab";
     cout << "Отсортированная строка: " << str << endl;
     //сортировка массива целых чисел
     int nums[] = {5, 7, 3, 9, 5, 1, 8};
     int i;
     shell(nums, 7);
     cout << "Отсортированный массив:";
     for(i=0; i<7; i++) cout << nums[i] << " ";
     return 0;
}
//Параметризованная функция сортировки методом Шелла
template <class Stype> void shell(Stype *item, int count);
{
     register int i, j, gap, k;
     Stype x;
     char a[5];
     a[0]=9; a[1]=5; a[2]=3; a[3]=2; a[4]=1;

     for (k=0; k<5; k++) {
          gap=a[k];
          for (i=gap; i<count; ++i) {
               x=item[i];
               for(j=i-gap; x<item[j] && j>=0; j=j-gap)
                    item[j+gap]=item[j];
               item[j+gap]=x;
          }
     }
}
```

Внутренний цикл **for** имеет два условия проверки. Очевидно, что сравнение **x<item[j]** необходимо по условиям процесса сортировки. Условие **j>=0** предотвращает выход за пределы массива **item**. Эти дополнительные проверки в некоторой степени снизят производительность алгоритма Шелла. Улучшенные версии алгоритма используют специальные элементы массива, называемые ***стражами*** (sentinels). Стражи не являются частью сортируемого массива. Они содержат завершающие элементы, обозначающие наибольший и наименьший возможные элементы массива. Это делает ненужной проверку границ массива, но требует конкретной информации о данных, что ограничивает общность функции сортировки.

Сортировка методом Шелла ставит целый ряд непростых математических задач, рассмотрение которых выходит за рамки данной книги. Поэтому читателю предлагается принять на веру тот факт, что время выполнения алгоритма пропорционально $n^{1.2}$ при сортировке n элементов. Это — существенный прогресс по сравнению с n-квадратными методами сортировки. Значимость этого улучшения ил

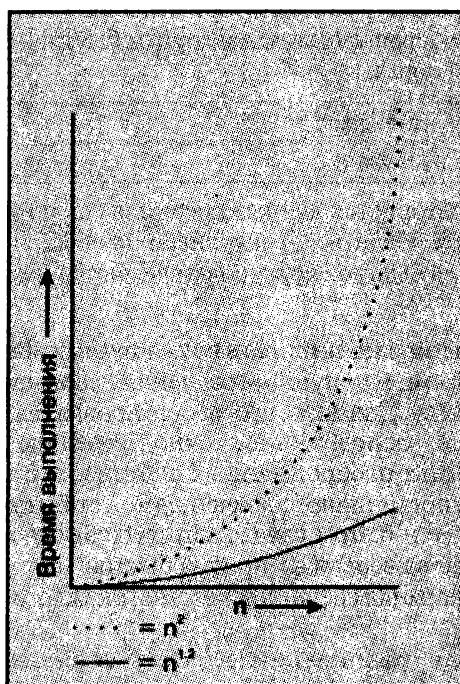


Рис. 1.3. Кривые n^2 и $n^{1.2}$

люстрируется рис. 1.3, где для сравнения приведены графики n^2 и $n^{1.2}$. Однако, следующий метод, который мы рассмотрим — метод быстрой сортировки — еще эффективнее метода Шелла.

Метод быстрой сортировки

Метод быстрой сортировки, разработанный и названный так его автором C.A.R. Hoare, по своим показателям превосходит все остальные алгоритмы, рассмотренные в этой книге. Более того, он считается наилучшим из разработанных на сегодняшний день универсальных алгоритмов. В его основе лежит метод перестановок — факт удивительный, особенно если иметь в виду плохие характеристики метода пузырьковой сортировки, также построенного на базе метода перестановок.

Алгоритм быстрой сортировки построен на основе идеи разбиения массива на разделы. Общая процедура заключается в выборе пограничного значения, называемого *компарандом* (comparand), которое разбивает сортируемый массив на две части. Все элементы, значение которых больше пограничного значения, переносятся в один раздел, а все элементы с меньшими значениями — в другой. Затем этот же процесс повторяется для каждой из частей и так до тех пор, пока массив не будет отсортирован. Например, допустим, что необходимо отсорти-

ровать следующий массив: "fedacb". Если в качестве компаранда использовать значение "d", то после первого прохода алгоритм быстрой сортировки упорядочит массив следующим образом:

Исходный массив	f e d a c b
Проход 1	b c a d e f

Затем этот процесс повторяется для каждого раздела, а именно, "bca" и "def". Процесс, как вы можете убедиться, рекурсивен по своей природе, и наилучшими реализациями метода быстрой сортировки являются именно рекурсивные алгоритмы.

Пограничное значение можно выбирать двумя путями. Во-первых, это можно делать случайным образом, или путем осреднения небольшого набора значений, принадлежащих к разделу. Для того, чтобы сортировка была оптимальной, следует выбирать значение, расположенное точно в середине диапазона значений. Однако, для большинства наборов данных добиться этого сложно. В наихудшем случае в качестве пограничного значения может быть выбран один из экстремумов. Однако, даже и в этом случае алгоритм быстрой сортировки все же работает. Ниже приведена версия алгоритма быстрой сортировки, которая выбирает в качестве компаранда срединный элемент каждого из разделов.

```
//Параметризованная реализация алгоритма быстрой сортировки
#include <iostream.h>
#include <string.h>

template <class Stype> void quick(Stype *item, int count);
template <class Stype> void qs(Stype *item, int left, int right);

main()
{
	//сортировка массива символов
	char str[] = "dcab";
	quick(str, (int)strlen(str));
	cout << "Отсортированная строка: " << str << endl;
	//сортировка массива целых чисел
	int nums[] = {5, 7, 3, 9, 5, 1, 8};
	int i;
	quick(nums, 7);
	cout << "Отсортированный массив:";
	for(i=0; i<7; i++) cout << nums[i] << " ";
	return 0;
}
//Входная функция быстрой сортировки
template <class Stype> void quick(Stype *item, int count);
{
	qs(item, o, count-1);
}
```

```
// Параметризованная функция быстрой сортировки
template <class Stype> void qs(Stype *item, int left, int
right)
{
     register int i, j;
     Stype x, y;

     i=left; j=right;
     x=item[(left+right)/2];

     do {
          while(item[i]<x && i<right) I++;
          while(x<item[j] && j>left) j—;

          if(i<=j) {
               y=item[i];
               item[i]=item[j];
               item[j]=y;
               i++; j—;
          }
     } while(i<=j);
     if(left<j) qs(item, left, j);
     if(i<right) qs(item, i, right);
}
```

В этой версии функция **quick( )** задает вызов основной сортирующей функции **qs( )**. Это позволяет поддерживать единообразный общий интерфейс **item** и **count**, но принципиально важным не является, так как функцию **qs( )** можно вызывать и непосредственно с использованием трех аргументов.

Получение количества сравнений и перестановок, которые выполняет алгоритм быстрой сортировки, требует математических выкладок, выходящих за рамки данной книги. Однако, среднее число выполняемых сравнений равно:

$n \log n$

Среднее количество перестановок приблизительно равно:

$n/6 \log n$

Эти числа существенно меньше, чем соответствующие показатели любого из ранее рассмотренных методов сортировки.

Вам следует иметь в виду один проблематичный аспект алгоритма быстрой сортировки. Если значение-компаранд для каждого из разделов является максимальным, то на практике метод быстрой сортировки вырождается в нечто, вполне достойное называться "медленной сортировкой" с временем выполнения, пропорциональным квадрату количества элементов. Однако, как правило, этого не происходит.

Необходимо тщательно выбирать метод определения значения компаранда. Чаще всего этот метод определяется сортируемыми данными. Например, при сортировке больших почтовых списков рассылки, где сортировка происходит по ZIP-коду, выбор довольно прост, так как ZIP-коды распределены довольно равномерно, и подходящий компаранд можно определить с помощью простой алгебраической функции. Однако, при работе с некоторыми базами данных, ключи сортировки могут совпадать или иметь настолько близкие значения, что наилучшим способом определения компаранда будет его случайный выбор. Широко распространенным и достаточно эффективным методом является выборка трех элементов из раздела и взятие их среднего.

Сравнение алгоритма быстрой сортировки со стандартной функцией qsort( )

Для того, чтобы в полной мере оценить преимущества использования шаблонов при построении универсальных функций по сравнению со старыми методами, откомпилируйте и запустите следующую программу. Она сортирует 10000 случайным образом сгенерированных целых, при этом сначала для этой цели используется стандартная (и уже устаревшая) библиотечная функция **qsort()**, а затем — только что рассмотренная параметризованная версия алгоритма быстрой сортировки. Вы наглядно убедитесь в том, что параметризованная функция выполняется почти в два раза быстрее!

```
//Сравнение функции qsort() и параметризованной версии быстрой сортировки
#include <iostream.h>
#include <string.h>
#include <time.h>

template <class Stype> void quick(Stype *item, int count);
template <class Stype> void qs(Stype *item, int left, int right);
int comp(const void *a, const void *b);

main()
{
     int nums1[10000], nums2[10000];
     int I;
     time_t start, end;

     for(i=0; i<10000; I++) nums1[i]=nums2[i]=rand();
     start=clock();
     quick(nums1, 10000);
     end=clock();
```

```
    cout<< "Время сортировки с использованием быстрой сортировки:
";
    cout<<end-start<<endl;

    start=clock();
    qsort(nums2, (unsigned) 10000, sizeof(int), comp);
    end=clock();

    cout<< "Время сортировки с помощью библиотечной функции qsort(): ";
    cout<<end-start<<endl;

    return 0;
}

//Входная функция быстрой сортировки
template <class Stype> void quick(Stype *item, int count);
{
    qs(item, o, count-1);
}
//Параметризованная функция быстрой сортировки
template <class Stype> void qs(Stype *item, int left, int right);
{
    register int i, j;
    Stype x, y;

    i=left; j=right;
    x=item[(left+right)/2];

    do {
        while(item[i]<x && i<right) I++;
        while(x<item[j] && j>left) j--;

        if(i<=j) {
            y=item[i];
            item[i]=item[j];
            item[j]=y;
            i++; j--;
        }
    } while(i<=j);
    if(left<j) qs(item, left, j);
    if(i<right) qs(item, i, right);
}

int comp(const void *a, const void *b)
{
    return *(int *) a  *(int *) b;
```

```
}
```

Как уже упоминалось ранее, причина того, что стандартная библиотечная функция сортировки работает медленнее параметризованной версии, заключается в том, что **qsort( )** использует существенное количество лишних операций при каждом ее вызове, так как ей необходимо передавать в качестве параметров размер сортируемых данных и адрес функции, осуществляющей сравнение. Поскольку чаще всего быстрая сортировка реализована как рекурсивная функция, эти параметры добавляют дополнительные операции не один раз, а многократно. Поскольку параметризованная функция-шаблон при каждой конкретизации создает версию, специфичную для каждого типа данных, то размер сортируемых данных известен, а операция сравнения встроена в функцию. Таким образом, необходимость в дополнительных параметрах отпадает, и дополнительные операции не увеличивают время выполнения.

Разница времени выполнения между двумя этими подходами доказывает, почему шаблоны фундаментальным образом изменяют лицо программирования.

Сортировка типов, определенных пользователем

Поскольку основной причиной для создания шаблонов является возможность его применения ко всем типам данных, возникает законный вопрос: как все рассмотренные нами методы сортировки можно применить к сортировке типов, определенных пользователем. Например, может возникнуть потребность выполнять сортировку классов. Рассмотрим следующий простой класс.

```
class address {
     char name[40];
     char street[40];
     char city[20];
     char state[3];
     char zip[11];
     // ...
};
```

Если вы попытаетесь выполнить сортировку массива объектов класса **address**, то получите ошибки компиляции, поскольку реляционные операторы типа **<or>**, используемые алгоритмами сортировки для сравнения элементов массива, не определены для объектов **address**. Таким образом, даже несмотря на то, что параметризованные алгоритмы сортировки могут сортировать любые типы данных, они не могут делать это успешно, если один или несколько операторов, используемых в теле функции, не определены для сортируемых типов данных. К счастью, эту проблему легко решить. Необходимо просто перегрузить необходимые операторы. Как только это будет сделано, параметризованные методы

алгоритмы сортировки смогут сортировать пользовательские типы с такой же легкостью, как и встроенные. Ясно, что определение необходимых реляционных операторов для типов классов, которые вы хотите сортировать, не представляет большого неудобства, поскольку, как правило, вам в любом случае потребуется перегружать несколько функций операторов при определении нового типа. Кроме того, перегрузка реляционных операторов обычно не является сложной задачей.

Сортировку пользовательских типов рассмотрим на примере нижеприведенной программы. Она использует параметризованный алгоритм быстрой сортировки для сортировки массива объектов **address**. В качестве ключа сортировки используется ZIP-код.

```
//Применение параметризованной быстрой сортировки к пользова-
тельским классам

#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

const int NUM_ITEMS=4;

class address {
     char name[40];
     char street[40];
     char city[20];
     char state[3];
     char zip[11];
public:

     address(char *n, char *s, char *c, char *st, char *z);
     address() {}; //null constructor

     int operator<(address &ob)
     {
          return strcmp(zip, ob.zip) < 0;
     }
     friend ostream &operator<<(ostream &stream, address &ob);
};

//Address constructor
address::address(char *n, char *s, char *c, char *st, char *z)

{
     strcpy(name, n);
     strcpy(street, s);
```

```
    strcpy(city, c);
    strcpy(state, st);
    strcpy(zip, z);
}

//Inserter for address class
ostream &operator<<(ostream &stream, address &ob)
{
    cout << ob.name << endl;
    cout << ob. street << endl;
    cout <<ob.city << " " << ob.state << " ";
    cout << ob.zip << endl << endl;
    return stream;
}
address addrs[NUM_ITEMS] = {
    address ("A. Alexander", "101 1st St", "Olney", "Ga", "55555"),
    address ("B. Bertrand", "22 12nd Ave", "Oakland", "Pa",
"34232"),
    address ("C. Carlisle", "33 3rd Blvd", "Ava", "Or", "92000"),
    address ("D. Dodger", "4 4rth Dr", "Fresno", "Mi", "45678"),
};

template <class Stype> void quick(Stype *item, int count);
template <class Stype> void qs(Stype *item, int left, int right);
main()
{
    int i;

    cout << "Unsorted Addresses:\n\n";
    for(i=0; i<NUM_ITEMS; i++) cout <<addrs[i];

    quick(addrs, NUM_ITEMS);
    cout << "Addresses sorted by ZIP code:\n\n";
    for(i=0; i<NUM_ITEMS; i++) cout <<addrs[i];

    return 0;
}

//Входная функция быстрой сортировки
template <class Stype> void quick(Stype *item, int count);
{
    qs(item, o, count-1);
}
//Параметризованная функция быстрой сортировки
```

```
template <class Stype> void qs(Stype *item, int left, int right);
{
     register int i, j;
     Stype x, y;
     i=left; j=right;
     x=item[(left+right)/2];

     do {
          while(item[i]<x && i<right) I++;
          while(x<item[j] && j>left) j—;

          if(i<=j) {
               y=item[i];
               item[i]=item[j];
               item[j]=y;
               i++; j—;
          }
     } while(i<=j);
     if(left<j) qs(item, left, j);
     if(i<right) qs(item, i, right);
}
```

Как видно из этого примера, класс **address** перегружает оператор <. Это — единственный реляционный оператор, используемый функцией **quick( )**. (Конечно, если вы хотите использовать другой алгоритм сортировки, вы можете перегружать другие реляционные операторы.) Перегрузка оператора < позволяет параметризованному алгоритму быстрой сортировки работать над объектами типа **address** без модификаций.

Вышеописанный метод может быть обобщен. Если вы хотите сортировать пользовательские типы, просто перегружайте необходимые реляционные операторы. После этого любой из параметризованных алгоритмов сортировки может использоваться без изменений.

Выбор метода сортировки

Каждый программист должен иметь в своем арсенале разнообразные методы сортировки, из которых можно выбрать нужный для каждой конкретной ситуации. Хотя для среднего случая оптимальным является метод быстрой сортировки, не всегда он будет наилучшим выбором. Например, при сортировке небольших списков (менее 100 элементов), временные издержки, вызванные рекурсивными вызовами быстрой сортировки, могут перевесить преимущества, даваемые этим совершенным алгоритмом. В этих редких случаях более простые алгоритмы — возможно, даже пузырьковый метод — могут оказаться более быстрыми. Кроме

того, другой метод следует предпочесть и в тех случаях, когда вы точно знаете, что сортируемый массив почти упорядочен, или же не хотите, чтобы при сортировке выполнялись перестановки элементов с одинаковыми ключами. Быстрая сортировка действительно является наилучшим универсальным алгоритмом, но это не означает, что в некоторых ситуациях вы не можете получить лучшего результата с использованием других подходов.

Поиск

Время от времени пользователю требуется найти запись в базе данных, используя ее ключ. Существуют метод поиска информации в неотсортированном массиве и метод поиска в отсортированном массиве. В состав стандартных библиотек всех компиляторов С++ входят функции поиска. Однако, как и в случае с сортировкой, эти универсальные функции обычно являются заимствованиями из С. Как следствие, в критичных ситуациях с высокими требованиями к производительности эти функции оказываются неэффективными из-за временных издержек, вызываемых лишними операциями процессора из-за передачи функции слишком большого числа параметров. В этом разделе мы рассмотрим две реализации параметризованных функций поиска.

Методы поиска

Нахождение информации в неотсортированном массиве требует последовательного поиска, начиная с первого элемента и завершая процесс или при нахождении совпадения, или при достижении конца массива. Если данные отсортированы, можно применить бинарный поиск, позволяющий быстрее найти нужные данные.

Последовательный поиск

Последовательный поиск реализуется легко. Ниже приведен пример программы, которая выполняет поиск в массиве указанной длины до нахождения совпадения с указанным ключом, используя параметризованную функцию поиска.

```
//Применение параметризованной функции последовательного
поиска
#include <iostream.h>

template <class Stype> int seq_search(Stype *item, int count,
Stype key);

main()
```

```
{
    char str[]="acbedfg";
    int nums[]={1, 7, 3, 5, 4, 6, 2};
    int index;

    index=seq_search(str, (int) sizeof(str), 'd');
    if(index>=0)
        cout<< "Найдено совпадение в позиции " << index << endl;
    else
        cout << "Совпадений не найдено\n";

    index=seq_search(nums, 7, 2);
    if(index>=0)
        cout<< "Найдено совпадение в позиции " << index << endl;
    else
        cout << "Совпадений не найдено\n";

    return 0;
}

//Параметризованный последовательный поиск
template <class Stype> int seq_search(Stype *item, int count, Stype key);
{
    register int t;

    for(t=0; t<count; ++t)
        if(key==item[t]) return t;
    return -1; //совпадения не найдено
}
```

Функция **seq_search( )** возвращает номер позиции совпадающего элемента, если он имеется, иначе она возвращает —1.

Очевидно, что последовательный поиск в среднем случае выполнит проверку 1/2*n* элементов. В наилучшем случае она выполнит проверку только одного элемента, а в наихудшем — *n* элементов. Если объем данных невелик, эта производительность будет приемлемой. Однако, при выполнении поиска в больших массивах, последовательный поиск будет чрезвычайно медленным. Конечно, если данные не отсортированы, у вас нет другого выбора, кроме последовательного поиска.

Бинарный поиск

Если данные, по которым требуется провести поиск, отсортированы, вы можете использовать несравненно более совершенный метод поиска. Это — ***бинарный поиск***, использующий метод "разделяй и властвуй". При использовании этого

метода на первом шаге проверяется срединный элемент. Если он больше ключа поиска, то проверяется срединный элемент первой половины массива; если он меньше ключа, то проверяется срединный элемент второй половины массива. Эта процедура повторяется до тех пор, пока не будет найдено совпадение, или до тех пор, пока больше не останется элементов, которые можно было бы проверить.

Например, допустим, что требуется найти число 4 в массиве:

1 2 3 4 5 6 7 8 9

Процедура бинарного поиска сначала выполнит проверку срединного элемента, который в данном случае равен 5. Так как это значение больше 4, бинарный поиск продолжается. Теперь будет выполнена проверка срединного элемента первой части массива, то есть:

1 2 3 4 5

Теперь срединный элемент равен 3. Так как он меньше 4, то первая половина этого интервала также будет отброшена. Бинарный поиск продолжится на интервале:

4 5

На этот раз совпадение будет найдено.

В наихудшем случае (максимальное количество шагов) бинарный поиск выполняет $\log_2 n$ сравнений.

В среднем случае количество сравнений будет несколько меньше. В наилучшем случае будет выполнено всего одно сравнение.

Нижеприведенная программа реализует параметризованную функцию бинарного поиска.

```
//Применение параметризованной функции бинарного поиска
#include <iostream.h>

template <class Stype> int binary_search(Stype *item, int count, Stype key);

main()
{
    char str[]="acbedfg";
    int nums[]={1, 7, 3, 5, 4, 6, 2};
    int index;

    index=binary_search(str, (int) sizeof(str), 'd');
    if(index>=0)
        cout<< "Найдено совпадение в позиции " << index << endl;
    else
        cout << "Совпадений не найдено\n";

    index=binary_search(nums, 7, 2);
```

```
    if(index>=0)
        cout<< "Найдено совпадение в позиции " << index << endl;
    else
        cout << "Совпадений не найдено\n";

    return 0;
}

//Параметризованный бинарный поиск
template <class Stype> int binary_search(Stype *item, int count, Stype key);
{
    int low, high, mid;

    low=0; high=count-1;
    while(low<=high) {
        mid=(low+high)/2;
        if(key<item[mid]) high=mid-1;
        else if(key>item[mid]) low=mid+1;
        else return mid; //совпадение
    }
    return -1; //совпадения не найдено
}
```

Рекомендации для самостоятельной разработки

Поскольку шаблоны еще относительно новы для C++, большинство программистов еще не в полной мере научились использовать их преимущества. Просмотрите свои наработанные библиотеки функций и выявите те из них, которые являются потенциальными кандидатами на параметризацию. Возможно, вы будете удивлены результатом — таких функций окажется очень много.

Поскольку конкретизированная версия шаблона настолько же эффективна, как и функция, написанная для конкретного типа данных, и намного эффективнее параметризованных функций в старом стиле, возможно, вам захочется просмотреть все такие функции с тем, чтобы переделать их как шаблоны. Следует помнить и о том, что прежний метод параметризации функций неизбежно приводил к дополнительным временным издержкам, так как передача функции дополнительных параметров влечет за собой дополнительные операции процессора. При использовании шаблонов эти временные издержки полностью устранены.

Наконец, вы можете построить новую библиотеку шаблонов от нуля. Поскольку шаблоны позволяют нам отделить метод от данных, такая библиотека, безусловно, найдет широкое применение.

Глава 2

Исследование параметризованных классов

В этой главе мы продолжим изучение шаблонов и исследуем параметризованные классы. Как упоминалось в главе 1, шаблон класса используется для построения родового класса. Создавая родовой класс, вы создаете целое семейство родственных классов, которые можно применять к любому типу данных. Таким образом, тип данных, которыми оперирует класс, указывается в качестве параметра при создании объекта, принадлежащего к этому классу. Как можно предположить, принципиальное преимущество параметризованного класса заключается в том, что он позволяет определять членов класса один раз, но применять класс к данным любых типов.

Наиболее широкое применение шаблоны классов находят при создании контейнерных классов. Фактически, как было указано в главе 1, создание контейнерных классов является одной из основных причин, по которым были введены в употребление шаблоны. Контейнерными классами в общем случае называются классы, в которых хранятся организованные данные. Например, массивы и связные списки являются контейнерами. Преимущество, даваемое определением параметризованных контейнерных классов, заключается в том, что как только логика, необходимая для поддержки контейнера, определена, он может применяться к любым типам данных без необходимости его переписывания. Благодаря этому, один раз написанный и отлаженный контейнерный класс можно использовать повторно. Например, параметризованный контейнер связного списка можно использовать для построения списков, содержащих почтовые адреса, заглавия книг или названия автомобильных запчастей. Поскольку контейнерные классы имеют исключительно важное значение, как практическое, так и историческое, мы рассмотрим возможности шаблонов классов на их примере.

В этой главе мы рассмотрим пять основных типов контейнеров:

- Ограниченный ("защищенный") массив
- Очередь
- Стек
- Связный список
- Бинарное дерево

Каждый из этих контейнеров выполняет конкретные операции сохранения и извлечения над заданной информацией и полученными запросами. В частности, все они могут сохранять и извлекать элемент (где элемент представляет собой одну информационную единицу). Все остальные разделы данной главы посвящены построению параметризованных версий этих контейнеров.

Обзор параметризованных классов

Прежде чем начинать строить параметризованные контейнерные классы, необходимо хотя бы обзорно ознакомиться с шаблонами классов. Определяя параметризованный класс, вы создаете класс, который определяет все алгоритмы, используемые этим классом, однако, фактический тип данных, над которыми производятся манипуляции, будет указан в качестве параметра при конкретизации объектов этого класса. Компилятор автоматически сгенерирует соответствующий объект на основании указанного вами типа.

Общая форма декларации параметризованного класса приведена ниже:

```
template <class Ttype> class class-name {
.
.
.
}
```

Здесь *Ttype* представляет собой имя типа шаблона, которое в каждом случае конкретизации будет замещаться фактическим типом данных. При необходимости вы можете определить более одного параметризованного типа данных, используя список с разделителем-запятой. В пределах определения класса имя *Ttype* можно использовать в любом месте.

Создав параметризованный класс, вы можете создать конкретную реализацию этого класса, используя следующую общую форму:

```
class-name <type> ob;
```

Здесь *type* представляет собой имя типа данных, над которыми фактически оперирует класс, и заменяет собой переменную *Ttype*.

Наконец, функции-члены параметризованного класса автоматически являются параметризованными. Их не обязательно декларировать как параметризованные с помощью ключевого слова **template**.

Ограниченные массивы

Простейшим примером контейнерного класса является ограниченный или "защищенный" массив. Как вы знаете, в С++ во время выполнения кода можно выйти за границу массива (или не дойти до нее) без генерации сообщений об ошибках времени выполнения. Хотя эта возможность позволяет С++ генерировать исключительно быстрый исполняемый код, но одновременно с этим она служит источником ошибок и доводила до исступления программистов еще с момента изобретения С++ (и его предшественника С). Однако, эту проблему можно успешно решить. Для этого необходимо создать класс, который содержит массив, и разрешить доступ к массиву только через перегруженный индексирующий оператор **[ ]**. В функции **operator[ ]( )** вы можете перехватывать индекс, выходящий за рамки диапазона массива. Поскольку механизм проверки границ будет одинаков для всех типов данных, имеет смысл создание параметризованного ограниченного массива, который вы сможете использовать каждый раз, когда вам потребуется "защищенный" массив. Как вы увидите далее, ограниченный параметризованный массив является простейшим, но, несмотря на это, одним из наиболее полезных контейнерных классов.

Как уже упоминалось, построение ограниченного параметризованного массива требует перегрузки оператора **[ ]**. Если вы еще не знакомы с перегрузкой этого оператора, вам поможет материал следующего раздела.

Перегрузка оператора [ ]

В С++ оператор **[ ]** при перегрузке считается бинарным оператором **[ ]** может быть перегружен только функцией-членом. Общая форма функции-члена **operator[ ]( )** приведена ниже:

```
type class-name::operator[](int index)
{
// ...
}
```

Чисто технически, параметр не обязательно должен принадлежать к типу **int**, однако, функции **operator[ ]( )**, как правило, используются для обеспечения индексации массивов, а для этого обычно используются целые значения. Как правило, функция **operator[ ]( )** возвращает значение того же типа, что и тип данных, хранящихся в индексируемом массиве.

Принцип работы оператора **[ ]** проще всего понять на примере **объекта с име**нем **ob**, проиндексированного следующим образом:

ob[9]

Индекс этого массива будет транслироваться в следующий вызов **функци** **operator[ ]( )**:

operator[](9)

Таким образом, выражение, используемое в качестве оператора **индексаци**, передается функции **operator[ ]( )** как явный параметр. Указатель **this буде**т указывать на **ob**, объект, сгенерировавший этот вызов.

Функцию **operator[ ]( )** можно разработать таким образом, чтобы **оператор** **]** можно было использовать как в левой, так и в правой части **оператор** присваивания. Для этого следует определить возвращаемое значение **функ**ции **operator[ ]( )** как ссылку и возвращать ссылку на указанный **элемен**т массива. В этом случае будут корректны следующие типы утверждений:

ob[9] = 10;

x = ob[9]

Построение параметризованного ограниченного массива

Нижеприведенная программа создает параметризованный ограниченный **мас**сив и иллюстрирует его использование. Обратите внимание, что **функци** **operator[]( )** возвращает ссылку, которая позволяет использовать ее в **любо**й части утверждения присваивания.

```
//Параметризованный ограниченный массив
#include <iostream.h>
#include <stdlib.h>

// Параметризованный класс ограниченного массива
template <class Atype> class atype {
     Atype *a;
     int length;
public:
atype(int size);
~atype() {delete [] a;}
Atype &operator[](int I);
};

//Конструктор для atype
template <class Atype> atype<AType>::atype(int size)
{
```

```
    register int i;

    length = size;
    a = new Atype[size]; // Динамическое выделение области хранения
    if(!a)    {
         cout << "Невозможно выделить массив.\n";
         exit(1);
    }
// Инициализация нулем
for(i = 0; i<size; I++) a[i] = 0;
}
//Обеспечение диапазона проверки для atype
template <class Atype> Atype &atype<AType>::operator[](int I)
{
    if(i<0 || i > length -1) {
         cout << "\nЗначение с индексом ";
         cout << i << " выходит за пределы диапазона. \n";
         exit (1);
    }
    return a[i];
}

main()
{
    atype<int> intob(20); //массив целых
    atype<double> doubleob(10); //массив переменных типа
    double
    int i;

    cout << "Массив целых: ";
    for (I = 0; I < 20; I ++) intob[i] = I;
    for (I = 0; I<20; I++) cout << intob[i] << " ";
    cout << endl;

    cout << "Double array: ";
    for (I = 0; I < 20; I ++) doubleob[i] = (doubleI) i *
    3.14;
    for (I = 0; I<20; I++) cout << doubleob[i] << " ";
    cout << endl;

    intob[45] = 100; // generates run-time error

    return 0;
}
```

Эта программа реализует параметризованный тип защищенного массива, а затем демонстрирует его использование, создавая массив целых и массив переменных типа **double**. (Можно попытаться создавать и массивы других типов.) Как показывает этот пример, параметризованные классы позволяют один раз написать и отладить код, который после этого можно использовать с данными любых типов без необходимости его дальнейшей переработки для каждого конкретного приложения.

Как видно из приведенного примера, принцип работы параметризованного класса массива прост. Пространство для массива выделяется динамически, а соответствующий указатель на этот адрес памяти хранится в переменной **a**. Размер массива передается в качестве параметра конструктору **atype** и хранится в переменной **length**. Таким образом, вы должны указывать размер массива при его создании. Поскольку создавать можно массивы всевозможных размеров, **atype** можно применять в самом широком диапазоне приложений. Однако, если вы заведомо знаете, что будете работать с массивами одного размера, рекомендуется использовать для **atype** массивы фиксированной длины, так как это несколько ускорит время выполнения.

Очереди

Очередь представляет собой линейный список, доступ к элементам которого осуществляется по принципу FIFO *(first-in, first-out)*. Таким образом, первым из очереди удаляется элемент, помещенный туда первым, затем — элемент, помещенный в очередь вторым, и так далее. Для очереди этот метод доступа и хранения является единственным. В отличие от массива, произвольный доступ к указанному элементу не допускается.

Очереди имеют чрезвычайно широкое распространение на практике. Вы можете увидеть их, например, в банке, в магазине, в ресторанчике быстрого питания и т. д. Кроме этих ситуаций из реальной жизни, очереди находят широкое применение и в программировании. В качестве этих примеров можно привести моделирование, диспетчеризацию задач операционной системой, а также буферизацию ввода/вывода.

Изучение принципов работы очереди начнем с рассмотрения двух функций: **qstore() и qretrieve()**. Функция **qstore()** помещает элемент в конец очереди, а функция **qretrieve()** извлекает из очереди первый элемент и возвращает его значение. В таблице 2.1 приведены результаты выполнения последовательности таких операций. Следует иметь в виду, что операция извлечения удаляет элемент из очереди, и хранившиеся в этом элементе данные будут разрушены, если только он не сохранен где-нибудь еще. Таким образом, если из очереди удалить все элементы, она окажется пустой.

Таблица 2.1. Очередь и принципы ее действия

Операция	Содержимое очереди
qstore (A)	A
qstore (B)	AB
qstore (C)	ABC
qretrieve( ) возвращает A	BC
qstore (D)	BCD
qretrieve( ) возвращает B	CD
qretrieve( ) возвращает C	D

Ниже приведен пример программы, создающей параметризованную очередь. Этот пример демонстрирует применение целых очередей и очередей с плавающей точкой, однако, на практике можно использовать данные любых типов.

```
//Параметризованный класс очереди
#include <iostream.h>
#include <stdlib.h>

// Параметризованный класс очереди
template <class Qtype> class queue {
     Qtype *q;
     int sloc, rloc;
     int length;
public:
     queue(int size);
     ~queue() {delete [] q;}
     void qstore(QType i);
     Qtype qretrieve();
};
template <class Qtype> queue<QType>::queue(int size)
{
     size++;

     q = new Qtype[size];
     if(!q)    {
          cout << "Невозможно создать очередь.\n";
          exit(1);
     }
     length = size;
     sloc = rloc = 0;
}
```

```
// Объект помещается в очередь
template <class Qtype> void queue<QType>::qstore(QType i)
{
     if(sloc+1==length) {
          cout << "Очередь заполнена.\n";
          return;
     }
     sloc++
     q[sloc] = i;
}

//Извлечение объекта из очереди
template <class Qtype> Qtype queue<QType>::qretrieve()
{
     if(rloc==sloc) {
          cout << "Очередь пуста.\n";
          return 0;
     }
     rloc++
     return q[rloc];
}

main()
{
     queue<int> a(5), b(5); //создаем две целых очереди

     a.qstore(100);
     b.qstore(200);

     a.qstore(300);
     b.qstore(400);

     cout << a.qretrieve() << " ";
     cout << a.qretrieve() << " ";
     cout << b.qretrieve() << " ";
     cout << b.qretrieve() << endl;

     queue<double> c(5), d(5); //создаем две очереди типа
double

     c.qstore(8.12);
     d.qstore(9.99);

     c.qstore(-2.00);
     d.qstore(0.986);
```

```
    cout << c.qretrieve() << " ";
    cout << c.qretrieve() << " ";
    cout << d.qretrieve() << " ";
    cout << d.qretrieve() << endl;

    return 0;
}
```

Ниже приведен образец вывода этой программы:

```
100   300   200   400
8.12  -2    9.99  0.986
```

Каждая очередь содержится в динамически выделяемом массиве, указатель на который содержит переменная **q**. Размер очереди передается как параметр конструктору класса **queue**. Этот размер хранится членом **length** структуры **queue**. Обратите внимание, что размер массива, выделяемого для поддержки очереди, превышает размер очереди. Причина этого в том, что в простейшей реализации очереди один элемент массива всегда пуст. Поэтому, если вы хотите построить очередь из 10 элементов, то массив, поддерживающий эту очередь, должен иметь фактическую длину 11 элементов.

Переменные **rloc** и **sloc** используются для индексации очереди, причем **sloc** содержит адрес, по которому будет сохранен следующий элемент, а **rloc** указывает индекс, определяющий следующий элемент, который будет извлечен. При каждом извлечении элемента из очереди **rloc** получает приращение. Таким образом, значение **rloc** приближается к **sloc**. Когда **rloc** и **sloc** содержат одно и то же значение, очередь пуста. Наконец, последнее замечание: несмотря на то, что операция **qretrieve( )** не разрушает информацию, извлекаемую из очереди, эту информацию можно считать удаленной, так как дальнейший доступ к ней будет невозможен.

Циклическая очередь

После внимательного изучения программы, приведенной в предыдущем примере, несложно догадаться о том, каким образом ее можно усовершенствовать. Эта программа останавливается после того, как будет достигнут предельный размер массива, используемого для хранения очереди. Вместо этого можно циклически возвращать индекс сохранения (**sloc**) и индекс извлечения (**rloc**) к началу массива. При такой реализации программы в очередь можно будет поместить любое количество элементов (при том условии, что элементы не только помещаются в очередь, но и извлекаются из нее). Такая реализация очереди называется *циклической очередью*, поскольку она использует массив, в котором хранятся элементы очереди, как циклический, а не линейный список.

Для создания циклической очереди необходимо изменить функции **qstore( )** **qretrieve( ),** как показано ниже:

```
//Параметризованный класс циклической очереди
#include <iostream.h>
#include <stdlib.h>

// Параметризованный класс циклической очереди
template <class Qtype> class queue {
     Qtype *q;
     int sloc, rloc;
     int length;
public:
     queue(int size);
     ~queue() {delete [] q;}
     void qstore(QType i);
     Qtype qretrieve();
};

template <class Qtype> queue<QType>::queue(int size)
{
     size++;

     q = new Qtype[size];
     if(!q)    {
          cout << "Невозможно создать очередь.\n";
          exit(1);
     }
     length = size;
     sloc = rloc = 0;
}

// Объект помещается в очередь
template <class Qtype> void queue<QType>::qstore(QType i)
{
     if(sloc+1==rloc || (sloc +1==length && !rloc)) {
          cout << "Очередь заполнена.\n";
          return;
     }
     q[sloc] = i;
     sloc++
     if(sloc=length) sloc = 0 // циклический переход
}
```

```
//Извлечение объекта из очереди
template <class Qtype> Qtype queue<QType>::qretrieve()
{
     if(rloc==length) rloc = 0; //циклический переход
     if(rloc==sloc) {
          cout << "Очередь пуста.\n";
          return 0;
     }
     rloc++
     return q[rloc - 1];
}

main()
{
     queue<int> a(10) //создаем целую очередь
     int i;

     //демонстрируем работу целой циклической очереди
     for (i=0; i<10; i++) a.qstore(i);
     cout << a.qretrieve() << endl;
     a.qstore(10);
     cout << a.qretrieve() << endl;
     a.qstore(11);
     cout << a.qretrieve() << endl;
     a.qstore(12);
     for(i=0; i<10; i++) cout <<a.qretrieve() << " ";
     cout << endl;

     queue<double> b(10); //создаем очередь типа double
     //демонстрируем работу целой циклической очереди
     for (i=0; i<10; i++) b.qstore((double) i*1.1);
     cout << b.qretrieve() << endl;
     b.qstore(10.0);
     cout << b.qretrieve() << endl;
     b.qstore(11.1);
     cout << b.qretrieve() << endl;
     b.qstore(12.2);
     for(i=0; i<10; i++) cout << b.qretrieve() << " ";

     return 0;
}
```

Ниже приведен вывод этой программы:

0

1

2

3 4 5 6 7 8 9 10 11 12

0

1.1

2.2

3.3 4.4 5.5 6.6 7.7 8.8 9.9 10 11.1 12.2

В этой версии очередь будет переполнена только в том случае, когда индекс извлечения будет на 1 больше, чем индекс сохранения. Очередь будет пуста, когда **rloc = sloc**. Во всех остальных случаях в очереди будет оставаться место для ввода еще одного элемента.

Стеки

Стек по смыслу противоположен очереди, так как использует противоположный по смыслу метод доступа, иногда называемый LIFO (*last-in, first-out*). Для того, чтобы получить представление о работе стека, представим себе стопку тарелок на столе. Первая тарелка, поставленная на стол, будет взята из стопки последней, а последняя — первой. Стеки широко применяются в системном программном обеспечении, включая компиляторы и интерпретаторы. Фактически компиляторы C++ используют стек для передачи аргументов функциям.

Две базовых операции со стеком — сохранение и извлечение — по традиции называются *push* и *pop* соответственно. Поэтому для того, чтобы реализовать стек, вам понадобятся две функции: **push()** (которая помещает элемент в стек) и **pop()** (которая извлекает элемент из стека). Кроме того, для реализации стека потребуется область памяти, которая будет использоваться в качестве стека. Для этой цели можно использовать массив, а можно выделить область памяти с помощью **new**. Функция извлечения, как и в случае с очередью, удаляет элемент из списка и уничтожает его содержимое (если только этот элемент не хранится где-либо еще). Пример работы стека приведен в таблице 2.2.

Приведенная ниже программа реализует параметризованный класс стека.

```
//Демонстрация параметризованного класса стека
#include <iostream.h>
#include <stdlib.h>
```

Таблица 2.2. Иллюстрация работы стека

Операция	Содержимое стека
push (A)	A
push (B)	BA
push (C)	CBA
pop( ) извлекает C	BA
push (F)	FBA
pop( ) извлекает F	BA
pop( ) извлекает B	A
pop( ) извлекает A	стек пуст

```
// Параметризованный класс стека
template <class Stype> class stack {
     Stype *stck;
     int tos;
     int length;
public:
     stack(int size);
     ~stack() {delete [] stck;}
     void push(SType i);
     Stype poph();
};

//Функция конструктора стека
template <class Stype> stack<SType>::stack(int size)
{
     stck = new Stype[size];

     if(!stck) {
          cout << "Невозможно создать стек.\n";
          exit(1);
     }
     length = size;
     tos = 0;
}

// Помещаем объект в стек
template <class Stype> void stack<SType>::push(SType I)
{
     if(tos==length) {
          cout << "Стек заполнен.\n";
```

```
        return;
    }
    stck[tos] = I;
    tos++;
}

//Извлекаем объект из стека
template <class Stype> Stype stack<SType>::pop()
{
    if(tos==0) {
        cout << "Стек пуст.\n";
        return 0;
    }
    tos—;
    return stck[tos];
}

main()
{
    stack<int> a(10); //Создаем целый стек
    stack<double> b(10); //Создаем стек типа double
    stack<char> c(10); //Создаем стек типа char

    int i;

// Используем стеки типа int и double
    a.push(10);
    b.push(100.1);
    a.push(20);
    b.push(10-3.3);

    cout << a.pop() << " ";
    cout << a.pop() << " ";
    cout << b.pop() << " ";
    cout << b.pop() << endl;

// Демонстрация символьного стека
    for (i=0; i<10; i++) c.push((char) 'A' + i);
    for (i=0; i<10; i++) cout << c.pop();
    cout << endl;

    return 0;
}
```

Ниже приведен вывод этой программы:

20 10 6.7 100.1

JIHGFEDCBA

Стек содержится в динамически выделяемом массиве, указателем на который является переменная **stck**. Размер стека передается конструктору класса стека как параметр. Этот размер содержится переменной-членом **length**. Переменная-член **tos** обозначает индекс следующей открытой позиции стека, которая является его вершиной. Если стек пуст, переменная **tos** равна нулю. Если **tos** больше индекса последнего сохранения, это указывает на то, что стек заполнен.

Применение стека отлично иллюстрирует калькулятор с четырьмя основными действиями. На настоящий момент большинство калькуляторов используют стандартную форму выражения, которая иногда называется инфиксной нотацией (*infix notation*) в форме *"операнд—оператор—операнд"*. Для того, чтобы получить сумму 100 и 200 с использованием этой нотации, необходимо сначала ввести первый операнд (**100**), затем — оператор (+), затем — второй операнд (**200**) и, наконец, нажать клавишу с изображением знака равенства (=). По контрасту с этой нотацией, многие ранние калькуляторы (и даже некоторые современные) используют так называемую постфиксную нотацию (*postfix notation*), при использовании которой сначала вводятся оба операнда, а затем — оператор. Для того, чтобы сложить 100 и 200 с использованием постфиксной нотации, необходимо сначала ввести 100, затем — 200, после чего нажать клавишу со знаком "+" (оператор). При этом операнды по мере их ввода помещаются в стек. Каждый раз, когда вводится оператор, из стека удаляются два операнда, а полученный в результате операции результат снова помещается в стек. Преимущество постфиксной нотации заключается в том, что она позволяет пользователю с легкостью вводить сколь угодно сложные и запутанные выражения. Нижеприведенный пример иллюстрирует использование стека при реализации постфиксного калькулятора.

Полный текст программы постфиксного калькулятора приведен ниже:

```
//Простой калькулятор, использующий параметризованный стек
#include <iostream.h>
#include <stdlib.h>

void calculator();

// Параметризованный класс стека
template <class Stype> class stack {
    Stype *stck;
    int tos;
    int length;
```

```
public:
     stack(int size);
     ~stack() {delete [] stck;}
     void push(SType i);
     Stype pop();
};

//Функция конструктора стека
template <class Stype> stack<SType>::stack(int size)
{
     stck = new Stype[size];

     if(!stck) {
          cout << "Невозможно создать стек.\n";
          exit(1);
     }
     length = size;
     tos = 0;
}

// Помещаем объект в стек
template <class Stype> void stack<SType>::push(SType I)
{
     if(tos==length) {
          cout << "Стек заполнен.\n";
          return;
     }
     stck[tos] = I;
     tos++;
}

//Извлекаем объект из стека
template <class Stype> Stype stack<SType>::pop()
{
     if(tos==0) {
          cout << "Стек пуст.\n";
          return 0;
     }
     tos—;
     return stck[tos];
}

main()
{
     calculator();
     return 0;
}
```

```
//Калькулятор с четырьмя действиями арифметики
void calculator()
{
    stack<double> calc(100);
    double a, b;
    char str[80];

    cout << "Простейший калькулятор\n";
    cout << "Для выхода введите 'q'\n";

    do {
        cout << ": ";
        cin >> str;
        switch(*str) {
            case '+':
                a=calc.pop();
                b=calc.pop();
                cout << a+b << endl;
                calc.push(a+b);
                break;
            case '-':
                a=calc.pop();
                b=calc.pop();
                cout << b-a << endl;
                calc.push(b-a);
                break;
            case '*':
                a=calc.pop();
                b=calc.pop();
                cout << a*b << endl;
                calc.push(a*b);
                break;
            case '/':
                a=calc.pop();
                b=calc.pop();
                if(!a) {
                    cout << "Деление на 0\n"
                    break;
                }
                cout << b/a << endl;
                calc.push(b/a);
                break;
            case '.': //Показать содержимое вершины стека
                a=calc.pop();
                calc.push(a);
```

```
                    cout << "Текущее значение в вершине стека: ";
                    cout << a << endl;
                    break;
                default:
                    calc.push(atof(str));
            }
        } while (*str != 'q');
}
```

Связные списки

Очереди и стеки имеют две общие черты. Во-первых, как очереди, так и стеки имеют очень строгие правила ссылок на хранящиеся в них данные, а именно: доступ к очереди определяется правилом FIFO, а для стека — правилом LIFO. Во-вторых, в обоих случаях операции извлечения данных по своей природе являются разрушающими. Иными словами, для того, чтобы получить доступ к элементу, хранящемуся в очереди или стеке, этот элемент необходимо удалить. Если элемент больше нигде не хранится, он будет удален. Следующий контейнерный класс, который мы изучим, свободен от этих ограничений. В отличие от стека или очереди связный список допускает гибкие методы доступа, поскольку каждый элемент данных содержит ссылку на следующий элемент данных в цепочке. Кроме того, для связных списков операция извлечения элемента не приводит к его удалению из списка и потере его данных. Для фактического удаления элемента связного списка требуется определять специальную операцию удаления. На материале данного раздела мы реализуем параметризованный класс связного списка.

Связные списки могут иметь одиночные или двойные связи. В списке с одиночными связями (*singly linked list*) каждый элемент содержит ссылку на следующий элемент данных. В списке с двойными связями (*doubly linked lists*) каждый элемент содержит ссылки на предыдущий и последующий элементы. Хотя списки с одиночными связями тоже не так уж и редки, все же наиболее широко распространены списки с двойными связями. Основную роль в этом играют следующие три основных фактора. Во-первых, список с двойными связями можно читать в обоих направлениях, и от начала к концу, и от конца к началу. Список с одиночными связями можно читать только в одном направлении. Во-вторых, поврежденный список с двойными связями проще перестраивать, так как с каждым из членов списка ассоциированы две ссылки. Наконец, некоторые типы операций над списками (например, удаление) проще выполняются над списками с двойными связками. В этой книге будут рассмотрены только списки с двойными связками, так как они обеспечивают наибольшую гибкость при сравнительно небольшом увеличении трудозатрат. Если вы заинтересуетесь и списками с одиночными связками, то найдете интересующую вас информацию в моей книге (*C: The Complete Reference, 3rd Ed.*, Osborne/McGraw-Hill.)

Возможно, вы уже знаете, что списки с двойными ссылками представляют собой динамические структуры данных, которые могут растягиваться или сжиматься в процессе исполнения вашей программы. Фактически принципиальным преимуществом динамической структуры данных является то, что она не должна иметь фиксированный размер на момент компиляции. Напротив, она может по мере надобности расширяться или сжиматься в процессе выполнения программы. Как уже говорилось, каждый объект в списке содержит ссылку на предыдущий и следующий объекты. При добавлении или удалении элементов ссылки соответствующим образом перестраиваются. Поскольку списки с двойными ссылками являются динамическими структурами данных, то чаще всего каждый из объектов списка является динамически выделяемым. Именно такой пример и рассматривается в настоящем разделе.

Каждый элемент, хранящийся в списке с двойными связями, состоит из трех частей, а именно: указателя на предыдущий элемент, указателя на следующий элемент и собственно информации, хранящейся в списке. Иллюстрация списка с двойными связями приведена на рис. 2.1.

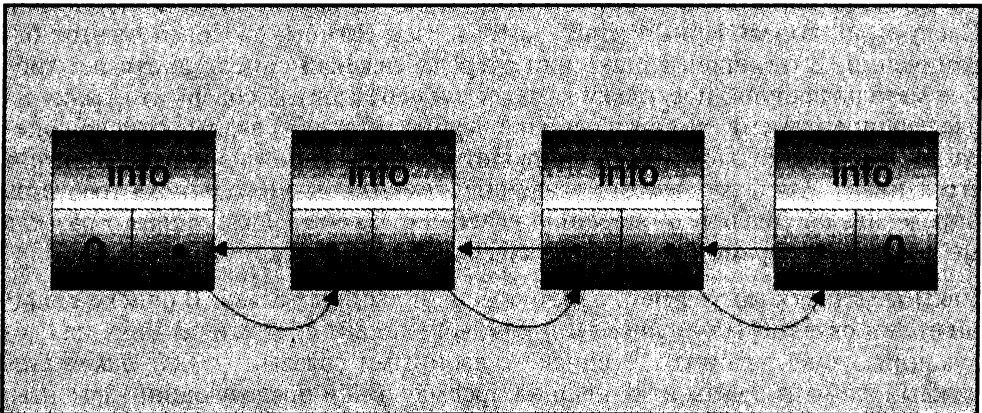


Рис. 2.1. Список с двойными связями.

Построение параметризованного класса списка с двойными связями

Для построения параметризованного списка с двойными связями используем простую иерархию классов. Один из классов, называемый **listob**, определяет природу объектов, которые будут храниться в списке. Этот класс затем наследуется другим классом, **dllist**, который фактически и реализует механизм списка с двойными связями.

Класс **listob** показан ниже:

```
template <class DataT> class listob {
public:
     DataT info; //информация
     listob<DataT> *next; //указатель на следующий объект
     listob<DataT> *prior; //указатель на предыдущий объект
     listob() {
          info = 0;
          next = NULL;
          prior = NULL;
     };
     listob(DataT c) {
          info = c;
          next = NULL;
          prior = NULL;
     }
     listob<DataT> *getnext() {return next;}
     listob<DataT> *getprior() {return prior;}
     void getinfo(DataT &c) {c = info;}
     void change(DataT c) {info =c;} //изменение элемента
     friend ostream &operator<<(ostream &stream,
     listob<DataT> o);
     friend ostream &operator<<(ostream &stream,listob<DataT> *o);
     friend istream &operator>>(istream &stream,listob<DataT> &o);
};

//Перегрузка << для объекта типа listob
template <class DataT>
ostream &operator<<(ostream &stream, listob<DataT> o)
{
     stream << o.info << endl;
     return stream;
}

//Перегрузка << для указателя на объект типа listob
template <class DataT>
ostream &operator<<(ostream &stream, listob<DataT> *o)
{
     stream << o->info << endl;
     return stream;
}

//Перегрузка >> для ссылки на объект типа listob
template <class DataT>
istream &operator>>(istream &stream, listob<DataT> &o)
```

```
{
    cout << "Введите информацию: ";
    stream << o.info;
    return stream;
}
```

.

Как видно из этого листинга, **listob** имеет три члена. Член **info** содержит информацию, хранящуюся в списке, определяемую родовым типом **DataT**. Указатель **next** будет указывать на следующий элемент списка, а указатель **prior** — на предыдущий. Обратите внимание на то, что в этом примере все члены объекта **listob** декларируются как **public**. Это сделано только в целях наглядности для упрощения демонстрации всех аспектов связного списка. При разработке собственных приложений программист может сделать их закрытыми (**private**) или защищенными (**protected**).

Кроме того, вместе с классом **listob** определен ряд операций, которые могут выполняться над объектами класса **listob**. В частности, можно извлечь и модифицировать информацию, ассоциированную с объектом, а также получить указатели на следующий и предыдущий элементы. Помимо этого, объекты типа **listob** могут вводиться и выводиться с помощью перегруженных операторов << и >>. Имейте в виду то, что операции, определенные внутри **listob**, независимы от механизма поддержки списка. **listob** определяет только природу данных, которые должны храниться в списке.

При построении каждого из объектов поля **prior** и **next** инициализируются значением NULL. Эти указатели сохранят значение NULL до тех пор, пока в список не будет введен первый элемент. Если включен инициализатор, то его значение будет скопировано в поле **info**, в противном случае значение **info** инициализируется нулем.

Функция **getnext( )** возвращает указатель на следующий элемент списка. Если достигнут конец списка, это значение будет равно NULL. Функция **getprior( )** возвращает указатель на предыдущий элемент списка в том случае, если он существует, в противном случае — значение NULL. В данном примере эти функции технически не являются необходимыми, так как указатели **next** и **prior** декларировались как **public**. Однако, они понадобятся, если сделать указатели **next** и **prior** закрытыми или защищенными.

Обратите внимание на то, что оператор << перегружается как для объектов типа **listob**, так и для указателей на объекты этого типа. Причина этого в том, что при использовании связных списков широко распространена практика получения доступа к элементам списка через указатель. Поэтому оператор << полезно перегружать, с тем чтобы он мог оперировать с переданным ему указателем на объект. Однако, поскольку нет причин полагать, что будет использоваться только этот метод, включена и вторая форма доступа, оперирующая непосредственно с самим объектом. Как альтернативу можно использовать и перегрузку << для использования ссылок на объект типа **listob**.

Хотя **listob** определяет природу списка с двойными ссылками, он сам не создает этого списка. Механизм построения связного списка реализуется классом **dllist**, приведенным ниже. Как можно видеть из приведенного листинга, он наследует **listob** и оперирует с объектами этого типа.

```
template <class DataT> class dllist : public listob<DataT> {
    listob<DataT> *start, *end;
public:
    dllist() {start = end = NULL;}
    ~dllist(); //деструктор для списка

    void store(DataT c);
    void remove(listob<DataT> *ob); //удаление элемента
    void frwdlist(); // отображение списка с начала
    void bkwdlist(); // отображение списка с конца

    listob<DataT> *find(DataT c); //указатель на найденное
совпадение

    listob<DataT> *getstart() {return start;}
    listob<DataT> *getend() {return end;}
};
```

Класс **dllist** поддерживает два указателя: один — на начало списка, а другой — на его конец. Оба эти указателя являются указателями на объекты типа **listob**. При создании списка оба указателя инициализируются значением NULL. Класс **dllist** поддерживает целый ряд операций над списком с двойными ссылками, в том числе:

- Ввод нового элемента в список
- Удаление элемента из списка
- Просмотр списка в любом направлении (от начала к концу или от конца к началу)
- Поиск в списке конкретного элемента
- Получение указателей на начало и конец списка
- Последующие разделы подробно рассматривают все эти процедуры.

Функция store( )

Добавление новых элементов в список производится с помощью функции **store()**. Пример реализации этой функции приведен ниже.

```
//Добавление нового элемента
template <class DataT> void dllist<DataT>::store(DataT c)
{
    listob<DataT> *p;

    p = new listob<DataT>;
    if(!p) {
        cout << "Ошибка выделения памяти.\n";
        exit(1);
    }

    p->info = c;

    if(start==NULL) {//первый элемент списка
        end = start = p;
    }
    else {
        p->prior = end;
        end->next = p;
        end = p;
    }
}
```

Для того, чтобы добавить в список новый элемент, необходимо создать новый объект типа **listob**. Поскольку связные списки представляют собой динамические структуры, для функции **store( )** логично будет получать объект динамически, используя **new**. После выделения памяти для объекта **listob** функция **store( )** присваивает информацию, передаваемую в переменной **c** члену **info** нового объекта, после чего добавляет объект в конец списка. Обратите внимание на то, что указатели **start** и **end** обновляются по мере надобности. Таким образом, **start** и **end** всегда будут указывать на начало и конец списка.

Поскольку объекты всегда добавляются в конец списка, список будет неотсортированным. Однако, при желании, вы можете модифицировать функцию **store( )** таким образом, чтобы она поддерживала отсортированный список.

Связный список, управляемый классом **dllist**, поддерживает список объектов типа **listob**. Тип данных, сохраняемых в объекте типа **listob**, не зависит от функции **store( )**.

Функция remove( )

Функция **remove( )** удаляет объект из списка. Листинг этой функции приведен ниже:

```
//Удаление элемента из списка с обновлением указателей на
начало и конец
template <class DataT> void
dllist<DataT>::remove(listob<DataT> *ob>
{
     if(ob->prior) { //не первый элемент
          ob->prior->next = ob->next;
          if(ob->next) // не последний элемент
               ob->next->prior = ob->prior;
          else // в противном случае, удаляется последний
          элемент
               end = ob->prior //обновление указателя на конец
               списка
          }
     else { //удаляется первый элемент списка
          if(ob->next) { //список не пуст
               ob->next->prior = NULL;
               start = ob->next;
          }
          else // теперь список пуст
               start = end = NULL;
     }
}
```

Функция **remove( )** удаляет из списка объект, на который указывает ее параметр **ob** (при этом параметр функции **ob** должен представлять собой допустимый указатель на объект типа **listob**). Удаляемый из списка объект может занимать одну из трех характерных позиций в списке, а именно — в начале, в конце или в середине списка (см. рис. 2.2). Функция **remove( )** обрабатывает все три ситуации.

Следует иметь в виду, что хотя функция **remove()** и удаляет объект из списка, этот объект физически не разрушается. Уничтожаются только ссылки, и объект просто "выпадает из цепочки". Разумеется, объект при необходимости можно и уничтожить. Для этого предназначается функция **delete( )**.

Как и функция **store( )**, функция **remove( )** никак не зависит от типа данных, фактически хранящихся в списке.

Отображение списка

Функции **frwdlist( )** и **bkwdlist( )** отображают содержимое списка в направлениях от начала к концу и обратно. Эти функции включены в пример в основном для того, чтобы нагляднее проиллюстрировать работу класса **dllist**. Кроме того, эти функции представляют собой довольно удобные средства отладки.

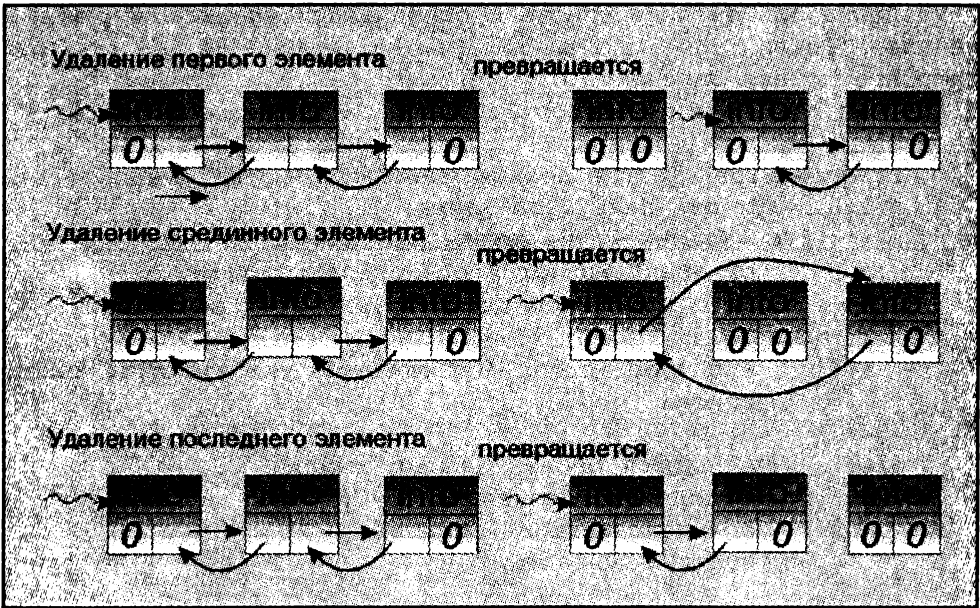


Рис. 2.2. Удаление объекта из списка с двойными ссылками

```
//Просмотр списка от начала к концу
template <class DataT> void dllist<DataT>::frwdlist()
{
     listob<DataT> *temp;

     temp = start;
     while(temp) {
          cout << temp->info << " ";
          temp = temp->getnext();
     }
     cout << endl;
}

// Просмотр списка в обратном направлении
template <class DataT> void dllist<DataT>::bkwdlist()
{
     listob<DataT> *temp;

     temp = end;
     while(temp) {
          cout << temp->info << " ";
          temp = temp->getprior();
     }
     cout << endl;
}
```

Поиск объекта в списке

Ниже приведена функция **find()**, которая возвращает указатель на объект списка, который содержит информацию, совпадающую с информацией, указанной параметром функции. Если совпадений не найдено, функция возвращает NULL.

```
//Поиск объекта, содержащего информацию, совпадающую с
указанной
template <class DataT> listob<Data> *dllist<Data
T>::find(DataT c)

{
     listob<DataT> *temp;

     temp = start;

     while(temp) {
        if(c==temp->info) return temp; // найдено совпадение
        temp = temp->getnext();
     }
     return NULL; // совпадений не найдено
}
```

Полный листинг параметризованного класса связного списка с двойными ссылками

В этом разделе приведено полное определение параметризованного класса связного списка с двойными ссылками и пример функции **main( )**, демонстрирующей его возможности. Обратите внимание на использование параметризованных данных при определении функций. Во всех случаях данные, над которыми осуществляются операции, обозначаются родовым типом **DataT**. Конкретизация типа данных осуществляется только в функции **main( )**.

```
// Параметризованный класс связного списка с двойными ссылками
#include<iostream.h>
#include<string.h>
#include<stdlib.h>

template <class DataT> class listob;

// Перегрузка << для объектов типа listob
template <class DataT>
ostream &operator<<(ostream &stream, listob<DataT> o)
```

```
{
     stream << o.info << endl;
     return stream;
}

// Перегрузка << для указателя на объект типа listob
template <class DataT>
ostream &operator<<(ostream &stream, listob<DataT> *o)
{
     stream << o->info << endl;
     return stream;
}

// Перегрузка >> для ссылки на объект типа listob
template <class DataT>
istream &operator>>(istream &stream, listob<DataT> &o)
{
     cout << "Введите информацию: ";
     stream << o.info;
     return stream;
}

// Параметризованный класс объекта списка
template <class DataT> class listob {
public:
     DataT info; //информация
     listob<DataT> *next; //указатель на следующий объект
     listob<DataT> *prior; //указатель на предыдущий объект
     listob() {
          info = 0;
          next = NULL;
          prior = NULL;
          };
     listob(DataT c) {
          info = c;
          next = NULL;
          prior = NULL;
          }
     listob<DataT> *getnext() {return next;}
     listob<DataT> *getprior() {return prior;}
     void getinfo(DataT &c) {c = info;}
     void change(DataT c) {info =c;} //изменение элемента
     friend ostream &operator<<(ostream &stream,
     listob<DataT> o);
     friend ostream &operator<<(ostream &stream,
     listob<DataT> *o);
```

```
    friend istream &operator>>(istream &stream,
    listob<DataT> &o);
};

// Параметризованный класс списка
template <class DataT> class dllist : public listob<DataT> {
    listob<DataT> *start, *end;
public:
    dllist() {start = end = NULL;}
    ~dllist(); //деструктор для списка

    void store(DataT c);
    void remove(listob<DataT> *ob); //удаление элемента
    void frwdlist(); // отображение списка с начала
    void bkwdlist(); // отображение списка с конца

    listob<DataT> *find(DataT c); //указатель на найденное
    совпадение

    listob<DataT> *getstart() {return start;}
    listob<DataT> *getend() {return end;}
};

// Деструктор dllist
template <class DataT> dllist<DataT>::~dllist()
{
    listob<DataT> *p, *p1;

    // освобождаем все элементы списка
    p = start;
    while(p) {
        p1 = p->next;
        delete p;
        p = p1;
    }
}

//Добавление нового элемента
template <class DataT> void dllist<DataT>::store(DataT c)
{
    listob<DataT> *p;

    p = new listob<DataT>;
    if(!p) {
        cout << "Ошибка выделения памяти.\n";
```

```
		exit(1);
	}

	p->info = c;

	if(start==NULL) {//первый элемент списка
		end = start = p;
	}
	else { //put on end
		p->prior = end;
		end->next = p;
		end = p;
	}
}

//Удаление элемента из списка с обновлением указателей на
 начало и конец
*/
template <class DataT> void
dllist<DataT>::remove(listob<DataT> *ob>
{
	if(ob->prior) { //не первый элемент
		ob->prior->next = ob->next;
		if(ob->next) // не последний элемент
			ob->next->prior = ob->prior;
		else // в противном случае, удаляется последний
		элемент
			end = ob->prior //обновление указателя на конец
			списка
		}
	else { //удаляется первый элемент списка
		if(ob->next) { //список не пуст
			ob->next->prior = NULL;
			start = ob->next;
		}
		else // теперь список пуст
			start = end = NULL;
	}
}

//Просмотр списка от начала к концу
template <class DataT> void dllist<DataT>::frwdlist()
{
	listob<DataT> *temp;
```

```
    temp = start;
    while(temp) {
        cout << temp->info << " ";
        temp = temp->getnext();
    }
    cout << endl;
}

// Просмотр списка в обратном направлении
template <class DataT> void dllist<DataT>::bkwdlist()
{
    listob<DataT> *temp;

    temp = end;
    while(temp) {
        cout << temp->info << " ";
        temp = temp->getprior();
    }
    cout << endl;
}

// Поиск объекта, содержащего информацию, совпадающую с
указанной
template <class DataT> listob<DataT>
*dllist<DataT>::find(DataT c)
{
    listob<DataT> *temp;
    temp = start;

    while(temp) {
        if(c==temp->info) return temp; // найдено
        совпадение
        temp = temp->getnext();
    }
    return NULL; // совпадений не найдено
}

main()
{
    // Демонстрация целого списка
    dllist<int> list;
    int i;
    listob<int> *p;
    list.store(1);
    list.store(2);
    list.store(3);
```

```
// Использование функций-членов для отображения списка
cout << "Список целых от начала к концу: ";
list.frwdlist();
cout << "Список целых от конца к началу: ";
list.bkwdlist();

cout << endl;

// "Ручной" просмотр списка
cout << "Ручной просмотр списка: ";
p = list.getstart();
while(p) {
     p->getinfo(i);
     cout << i << " ";
     p = p->getnext(); //следующий элемент
}

cout << endl << endl;

// Поиск элемента
cout << "Поиск 2\n";
p = list.find(2);
if(p) {
     p->getinfo(i);
     cout << "Найден элемент " << i << endl;
}

cout << endl;

// Удаление элемента
p->getinfo(i);
cout << "Удаление элемента " << i << ".\n";
list.remove(p);
cout << "Список после удаления: ";
list.frwdlist();

cout << endl;

// Добавление нового элемента
cout << "Добавление нового элемента.\n";
list.store(4);
cout << "Список после добавления: ";
list.frwdlist();
```

```
cout << endl;
//Изменение информации
p = list.find(1);
if(!p) {
     cout << "Ошибка, элемент не найден.\n";
     return 1;
}

p->getinfo(i);
cout << "Изменение " << i << " на 5.\n";
p->change(5);
cout << "Список после изменения: ";
list.frwdlist();
cout << "Список в обратном порядке: ";
list.bkwdlist();

cout << endl;

// Демонстрация << и >>
cin >> *p;
cout << " Теперь элемент содержит: " << *p;

cout << "Список от начала к концу: ";
list.frwdlist();
cout << endl;

// Удалим начальный элемент списка
cout << "После удаления начального элемента: ";
p = list.getstart();
list.remove(p);
list.frwdlist();

cout << endl;

// Удалим конечный элемент списка
cout << "После удаления конечного элемента: ";
p = list.getend();
list.remove(p);
list.frwdlist();

cout << endl;

// Построение списка типа double
dllist<double> dlist;
```

```
    dlist.store(98.6);
    dlist.store(212.0);
    dlist.store(88.9);

    // Использование функций-членов для отображения списка
    cout << "Список элементов типа double от начала к концу
    cout << "и от конца к началу.\n";
    dlist.frwdlist();
    dlist.bkwdlist();

    return 0;
}
```

Отработав, эта программа даст следующий вывод:

```
Список целых от начала к концу: 1    2    3
Список целых от конца к началу: 3    2    1

Ручной просмотр списка: 1  2  3

Поиск 2
Найден элемент 2

Удаление 2
Список после удаления: 1  3

Добавление нового элемента.
Список после добавления: 1  3  4

Изменение 1 на 5.
Список после изменения: 5  3  4
Список в обратном порядке: 4    3    5

Введите информацию: 10
Теперь элемент содержит: 10
Список от начала к концу: 10  3  4

После удаления начального элемента: 3    4

После удаления конечного элемента: 3

Список элементов типа double от начала к концу и от конца к началу:
98.6    212  88.9
88.9    212  98.6
```

Этот пример создает связный список с двойными ссылками и демонстрирует различные функции, при помощи которых с этим списком можно оперировать. После этого программа строит такой же список объектов типа **double**, с тем чтобы продемонстрировать родовые свойства класса связных списков. Попытайтесь самостоятельно создать разнообразные списки, в которых хранятся данные различных типов. В списке можно хранить любые, сколь угодно сложные составные структуры, например, почтовые адреса.

Бинарные деревья

Наконец, последней структурой данных, которую мы изучим в данной главе, является бинарное дерево. Существует множество типов деревьев, однако, мы выбираем для изучения именно бинарные, по той причине, что если бинарное дерево отсортировано, оно позволяет быстро осуществлять поиск, вставку и удаление элементов. Каждый элемент дерева содержит собственно информацию, а также ссылки на члены, расположенные левее и правее данного элемента. Пример небольшого дерева приведен на рис. 2.3.

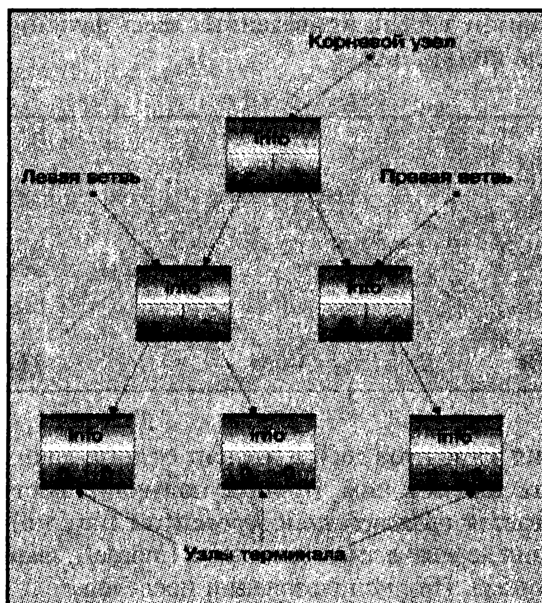
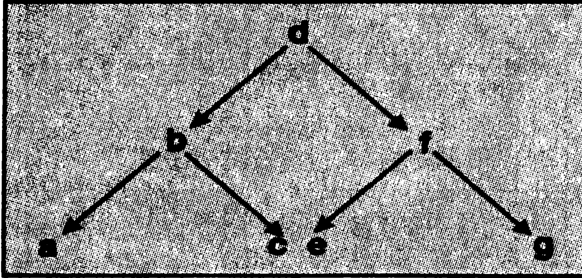


Рис. 2.3. Пример бинарного дерева с тремя уровнями

При обсуждении деревьев необходимо условиться о терминологии. Программисты не славятся блестящим знанием грамматики или удачно изобретенными терминами, и терминология в отношении деревьев - характерный тому пример. *Корнем* (root) называется первый элемент дерева. Каждый элемент данных называется *узлом* (node), а иногда - *листом* (leaf) дерева. Фрагмент дерева называется *поддеревом* (subtree) или *ветвью*. Узел, не имеющий отходящих от него ветвей или поддеревьев, называется *терминальным* или *конечным узлом* (terminal node). *Высота дерева* (height) определяется количеством уровней, на которых располагаются его узлы. Работая с деревьями, можно представлять их себе по схеме, которую вы вычертите на бумаге. Однако, не следует забывать о том, что деревья - это только метод логической организации информации в памяти, а память линейна.

В этом смысле бинарное дерево представляет собой разновидность связного списка. Элементы дерева можно добавлять, удалять, а также получать к ним доступ в произвольном порядке. Кроме того, операция извлечения является неразрушающей. Хотя деревья довольно просто изобразить в виде схем, они представляют целый ряд проблем, сложных для программирования. Исследование, представленное в этой книге, поднимает только некоторые из них.

Большинство из функций, используемых деревьями, являются рекурсивными, так как деревья представляют собой рекурсивную структуру данных. Каждое поддерево также является деревом. Таким образом, все процедуры, разработанные в процессе данного обсуждения, будут рекурсивными. Существуют и нерекурсивные версии приведенных здесь функций, однако, их код гораздо сложнее для понимания.



Порядок следования элементов дерева зависит от того, каким образом планируется организовать доступ к ним. Процесс получения доступа к каждому из узлов дерева называется *прохождением дерева* (tree traversal). Существует три способа прохождения дерева: *последовательный* (inorder), *нисходящий* (preorder) и *восходящий* (postorder). При использовании последовательного метода дерево проходится, начиная с левого поддерева вверх к корню, затем к правому поддереву. При нисходящем методе сначала проходится корень, затем — левое поддерево, и, наконец — правое. При восходящем методе прохождение начинается с левого поддерева, затем — правого с переходом к корню. Порядок прохождения приведенного на иллюстрации дерева при каждом из трех методов будет следующим:

inorder	a b c d e f g
preorder	d b a c f e g
postorder	a c b e g f d

Хотя дерево не обязательно должно быть отсортировано, большинство практических приложений требуют этого. Разумеется, порядок сортировки дерева будет определяться методом, которым это дерево должно проходиться. Все последующие разделы данной главы подразумевают последовательный метод

прохождения дерева (***inorder***). Таким образом, отсортированное бинарное дево представляет собой дерево, левая ветвь которого содержит узлы, значения которых меньше корневого, а правая — узлы, имеющие значения больше корневого.

Теперь, условившись о терминологии, построим параметризованный класс бинарного дерева. В целях экономии бумаги создадим самый простой класс (минимально необходимый для понимания сути вопроса). Поняв основные принципы, вы сможете самостоятельно усложнить этот класс и расширить его возможности. Ряд полезных идей, которые вы можете при этом реализовать, приведен в конце этой главы.

Параметризованный класс дерева

Для построения параметризованного класса бинарного дерева будет использоваться следующий шаблон с названием **tree**:

```
template <class DataT> class tree {
     DataT info;
     tree *left;
     tree *right;
public:
     tree *root;
     tree() {root = NULL;}
     void stree(tree *r, tree *previous, DataT info);
     tree *dtree(tree *r, DataT key);
     void preorder(tree *r);
     void inorder(tree *r);
     void postorder(tree *r);
     void print_tree(tree *r, int l);
     tree *seach_tree(tree *r, DataT key);
}
```

Каждый узел дерева будет представлять собой объект типа **tree**. Член **info** будет содержать информацию, хранящуюся в каждом из узлов. Для каждого из объектов **left** будет представлять собой указатель на левое поддерево, **right** - указатель на правое поддерево, а **root** — указатель на корень всего дерева. Обратите внимание - указатель **root** не является необходимым (указатель на корень дерева можно хранить в любом другом месте), но включен для удобства. Его наличие позволяет определить начало всего дерева, находясь в любом из его узлов. В некоторых случаях эта возможность может оказаться очень удобной.

Добавление элементов в дерево

Для включения в состав дерева новых элементов используется функция **stree()**, приведенная ниже:

```
template <class DataT> void tree<DataT> ::stree(
    tree *r, tree *previous, DataT info)

{
    if(!r) {
        r = new tree;
        if(!r) {
            cout << "Недостаточно памяти.\n";
            exit(1);
        }
        r->left = NULL;
        r->right = NULL;
        r->info = info;
        if(!root) root =r; //первый элемент
        else {
            if(info < previous->info) previous->left = r;
            else previous->right = r;
        }
        return;
    }

    if(info < r->info)
        stree(r->left, r, info);
    else
        stree(r->right, r, info)
}
```

Эта функция вставляет объект в бинарное дерево, отслеживая ссылки на элементы дерева и перемещаясь влево или вправо, в зависимости от содержимого поля **info**, до тех пор, пока для элемента не будет найдено соответствующее ему место в иерархии дерева. Функция **stree( )** представляет собой рекурсивный алгоритм, как и большинство процедур, связанных с деревьями. Если вы попытаетесь создать аналогичную процедуру с использованием итеративных методов, она получится в несколько раз длиннее. При вызове функции следует указывать следующие аргументы: указатель на корень дерева или поддерева, в котором должен производиться поиск, указатель на предыдущий узел, сохраняемая информация. При первом вызове второй параметр может иметь значение NULL.

Фактически эта функция сортирует передаваемую ей информацию прежде, чем поместить ее в дерево. Она представляет собой вариацию алгоритма сортировки методом вставки, обсуждавшегося в предыдущей главе. В общем случае ее производительность достаточно хороша, хотя метод быстрой сортировки все же является наилучшим из универсальных методов, так как требует меньше памяти и генерирует меньше избыточных операций процессора. Тем не менее, если вам требуется строить дерево "от нуля" или поддерживать уже упорядоченное дерево, рекомендуется всегда добавлять новые элементы с помощью функции **stree( )**, одновременно со вставкой элементов осуществляющей их сортировку.

Прохождение дерева

Для прохождения дерева, построенного с помощью функции **stree( )**, с последующей распечаткой поля **info** каждого из узлов этого дерева можно использовать нижеприведенную функцию **inorder( )**:

```
template <class DataT> void tree<DataT> ::inorder(tree *r)
{
    if(!r) return;

    inorder(r->left);
    if(r-info) cout << r->info << " ";
    inorder(r->right);
}
```

Эту функцию следует вызывать, указывая в качестве аргумента указатель на корень поддерева, которое требуется пройти. Если требуется пройти все дерево, вызовите функцию, указав аргументом указатель на корень дерева. Выход из этой рекурсивной функции происходит, когда в процессе прохождения дерева встречается терминальный узел (нулевой указатель).

Ниже приведены соответствующие функции для прохождения дерева в нисходящем и восходящем порядках.

```
template <class DataT> void tree<DataT> ::preorder(tree *r)
{
    if(!r) return;

    if(r->info) cout << r->info << " ";
    preorder(r->left);
    preorder(r->right);
}
template <class DataT> void tree<DataT> ::postorder(tree *r)
```

```
{
     if(!r) return;

     postorder(r->left);
     postorder(r->right);
     if(r->info) cout << r->info << " ";
}
```

Теперь рассмотрим короткую, но интересную программу, которая строит отсортированное бинарное дерево, а затем обходит его в последовательном порядке и выводит на экран. Для того, чтобы программа могла осуществлять вывод на экран, требуется лишь незначительная модификация функции **inorder()**. Для того, чтобы распечатанное дерево правильно выглядело на экране, правое поддерево должно распечатываться раньше левого. (Технически это — противоположность последовательному прохождению дерева.) Эта новая функция называется **print_tree( )** и приведена ниже:

```
template <class DataT> void tree<DataT> ::print_tree(tree *r,
int l)
{
     int i;

     if(!r) return;

     print_tree(r->right, l+1);
     for(i=0, i<l, ++i) cout << " ";
     cout << r->info << endl;
     print_tree(r->left, l+1);
}
```

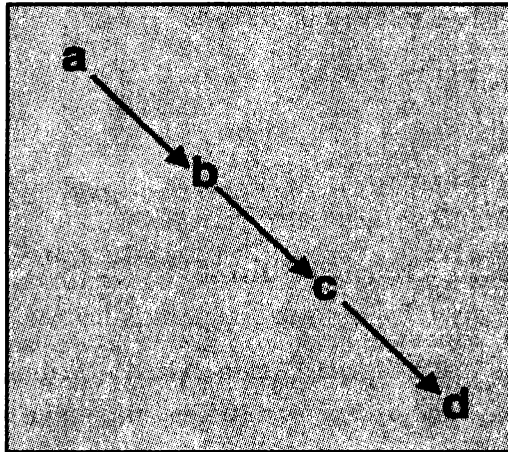
Ниже приведен полный листинг программы отображения дерева. Для того, чтобы хорошо понять принципы ее работы, попробуйте вводить разнообразные деревья.

```
// Программа вывода бинарного дерева
#include <iostream.h>
#include <stdlib.h>
template <class DataT> class tree {
     DataT info;
     tree *left;
     tree *right;
public:
     tree *root;
     tree() {root = NULL;}
```

```
    void stree(tree *r, tree *previous, DataT info);
    void print_tree(tree *r, int l);
};

template <class DataT> void tree<DataT> ::stree(
    tree *r, tree *previous, DataT info)
{
    if(!r) {
        r = new tree;
        if(!r) {
            cout << "Недостаточно памяти.\n";
            exit(1);
        }
        r->left = NULL;
        r->right = NULL;
        r->info = info;
        if(!root) root =r; //первый элемент
        else {
            if(info < previous->info) previous->left = r;
            else previous->right = r;
        }
        return;
    }

    if(info < r->info)
        stree(r->left, r, info);
    else
        stree(r->right, r, info)
}
// Отображение дерева
template <class DataT> void tree<DataT> ::print_tree(tree *r,
int l)
{
    int i;

    if(!r) return;

    print_tree(r->right, l+1);
    for(i=0, i<l, ++i) cout << " ";
    cout << r->info << endl;
    print_tree(r->left, l+1);
}

main()
{
```

```
    char s[80];
    tree<char> chTree;
    do {
        cout << "Введите символ (точку - для останова): ";
        cin >> s;
        if(*s!='.') chTree.stree(chTree.root, NULL, *s);
        } while (*s!='.');

        chTree.print_tree(chTree.root, 0);

        return 0;
}
```

Несколько раз выполнив эту программу, вы заметите, что некоторые деревья *сбалансированы* — каждое их поддерево имеет такую же или почти такую же высоту, как и любое другое, в то время как другие деревья чрезвычайно далеки от этого. Фактически, если вы введете следующее дерево: **a b c d**, оно будет выглядеть примерно так же, как на нижеприведенной иллюстрации:



Никаких левых поддеревьев в этом случае не появится. Такое дерево называется *вырожденным* (degenerate tree), так как фактически оно выродилось в линейный список. В среднем случае, если порядок данных, которые вы используете как входные данные для построения бинарного дерева, достаточно произволен, получится дерево, приближающееся к сбалансированному. Если информация уже отсортирована и данные упорядочены, результатом операции построения дерева будет вырожденное дерево. (Если требуется сохранять сбалансированное дерево, его можно переупорядочивать при каждом добавлении нового элемента. Однако, этот процесс сложен и лежит за пределами материалов, обсуждаемых в этой главе.)

Поиск по дереву

Операции поиска для бинарных деревьев реализуются легко. Ниже приведен листинг функции, которая возвращает указатель на узел дерева, который содержит информацию, совпадающую с ключом поиска. Если искомый элемент не будет найден, функция возвратит NULL.

```
template <class DataT>
tree<DataT> *tree<DataT>::search_tree(tree *r, DataT key)
{
    if(!r) return r; // пустое дерево
    while(r->info !=key) {
        if(key < r->info) r = r->left;
        else r = r->right;
        if(r==NULL) break;
    }
    return r;
}
```

Эту функцию следует вызывать, указывая в качестве аргументов указатель на начало поддерева, в котором требуется произвести поиск, и искомую информацию. Для того, чтобы выполнить поиск по всему дереву, передайте функции указатель на корневой узел дерева.

Удаление элемента дерева

К сожалению, удаление узла из дерева не столь простой процесс, как поиск по дереву. Удаляемый узел может быть корневым, левым или правым. Кроме того, узел может давать начало другим поддеревьям (их может быть от 0 до 2). Процесс переприсваивания значений указателям представляет собой рекурсивный алгоритм, приведенный ниже.

```
template <class DataT>
tree<DataT> *tree<DataT>::dtree(tree *r, DataT key)
{
    tree *p, *p2;

    if(!r) return r; // элемент не найден

    if(r->info==key) { // delete root
        //this means an empty tree
        if(r->left==r->right) {
            free(r);
            if(r==root) root = NULL;
            return NULL;
        }
```

```
        else if(r->left==NULL) {
            p = r->right;
            free(r);
            if(r==root) root = p;
            return p;
        }
        else if(r->right==NULL) {
            p = r->left;
            free(r);
            if(r==root) root = p;
            return p;
        }
        else {
            p2 = r->right;
            p = r->right;
            while(p->left) p = p->left;
            p->left = r->left;
            free(r);
            if(r==root) root = p2;
            return p2;
        }
    }
    if(r->info<key) r->right = dtree(r->right, key);
    else r->left = dtree(r->left,key);
    return r;
}
```

Эту функцию необходимо вызывать с указателем на начало (корень) поддерева, по которому требуется выполнить поиск информации, которую нужно удалить. Для того, чтобы выполнить поиск по всему дереву, необходимо передать функции указатель на корневой узел дерева.

Бинарные деревья представляют собой необычайно мощный, гибкий и эффективный инструмент. Поскольку при поиске по сбалансированному бинарному дереву (наихудший случай) выполняется $\log_2 n$ сравнений, оно по сравнению со связным списком представляет собой структуру, гораздо более удобную для поиска информации.

Рекомендации для самостоятельной разработки

В первую очередь попробуйте использовать описанные в этой главе контейнерные классы для хранения пользовательских типов данных. Например, поэкспериментируйте с построением собственной программы построения поч-

товых списков рассылки. Например, постройте класс почтовых списков, который будет содержать информацию об имени и адресе получателя, и перегружать необходимые операторы. Попробуйте хранить почтовый список в виде связного списка и в виде бинарного дерева.

Как уже говорилось ранее, класс **tree** в приведенной для примера программе был намеренно сделан максимально упрощенным. Попробуйте развить класс бинарных деревьев в соответствии со своими потребностями и расширить его новыми возможностями. Во-первых, по аналогии с классом связных списков можно попробовать отделить объекты дерева от операций, выполняемых над деревом. Для этого можно, например, определить класс **treeob**, который будет определять природу объектов, формирующих дерево, а затем построить класс **tree**, наследующий класс **treeob**. Кроме того, можно перегружать операторы ввода и вывода по отношению к классам дерева.

Все пять контейнеров, которые были исследованы в этой главе, являются достаточно простыми и широко распространенными. Однако, существуют и другие типы контейнеров. Попробуйте создать некоторые типы самостоятельно. Для самостоятельной реализации предлагаем вам несколько идей:

- Постройте приоритетную очередь. При добавлении элемента в такую очередь порядковый номер нового элемента должен определяться его приоритетом.
- Постройте ассоциативный связный список, в котором элементы содержат ссылки на соответствующие элементы
- Постройте мультивариантное дерево (например, дерево Байера) в отличие от бинарного дерева.

Благодаря новым возможностям, которые предоставляют параметризованные классы, усилия, затраченные вами на выполнение этих упражнений, многократно окупятся на протяжении всей вашей карьеры программиста.

Глава 3

Объектно-ориентированная программа разбора математических выражений

В этой главе рассматривается один из наиболее таинственных процессов всей науки программирования: разбор выражений. Прочтя эту главу, вы сможете самостоятельно создать объектно-ориентированную программу разбора выражений. Программы этого рода используются, например, для выполнения оценки алгебраических выражений, таких, как (10 - 8)*3. Кроме того, они чрезвычайно полезны для различных приложений из самых разных областей. Наконец, они представляют собой одну из интереснейших и труднейших тем программирования. По различным причинам, процедуры, используемые для построения программ, осуществляющих разбор выражений, не слишком широко распространены, и о них довольно мало информации. Поэтому многие подкованные и эрудированные программисты, тем не менее, чувствуют себя озадаченными, сталкиваясь с проблемой разбора выражений.

Загадочным разбор выражений кажется только на первый взгляд, а в действительности эта задача оказывается достаточно ясной и даже по многим параметрам — не более сложной, чем многие другие задачи программирования. Задача четко определена и легко поддается формализации, так как заложенные в ее основу принципы являются ничем иным, как хорошо всем известными жесткими законами алгебры. В этой главе мы проанализируем алгоритм, известный под названием *рекурсивно-нисходящего алгоритма разбора выражений* (recursive-descent parser), а также все процедуры, необходимые для его поддержки, которые позволяют выполнять оценку сложных численных выражений. Мы создадим несколько версий этого алгоритма. Первые две версии — непа-

раметризованные, в то время как третья представляет собой параметризованный алгоритм, применимый к любым числовым типам. Однако, прежде, чем приступать к разработке любой, самой простейшей программы разбора выражений, необходимо хотя бы обзорно ознакомиться с теоретическими основами выражений и их разбора.

Выражения

Поскольку программа разбора выражений осуществляет оценку алгебраического выражения, важно понимать, из каких составных частей это выражение строится. Хотя в общем случае выражение можно построить из информации любых типов, в этой главе мы ограничимся обсуждением только числовых выражений. Основой для построения числовых выражений служат следующие элементы:

- Числа
- Операторы(+, -, /, *, ^, %, =)
- Скобки
- Переменные

Оператор ^ обозначает возведение в степень, как в языке BASIC, а = представляет собой оператор присваивания. В выражениях все эти элементы могут комбинироваться в любых сочетания, допускаемых законами алгебры. Ниже приведен ряд примеров допустимых выражений:

10 - 8

(100 - 5)*14/6

a + b - c

10^5

a = 10 - b

Приоритеты операторов определяются следующим образом:

высший	+ - (унарные)
	^
	* / %
	+ -
низший	=

Операторы с одинаковыми приоритетами выполняются в порядке их следования в направлении слева направо.

В примерах данной главы все переменные являются одиночными буквами (иными словами, для применения доступны 26 переменных, от **A** до **Z**). Переменные не чувствительны к регистру (это означает, что **A** и **a** трактуются как одна и та же переменная). В первой версии программы все численные значения оцениваются как имеющие тип **double** (вы с легкостью можете написать программы, которые будут обрабатывать другие типы значений). Наконец, в эту версию включены только минимальные возможности по обработке ошибок. Это сделано, с тем чтобы не перегружать программу, четко отследить ее логику и сделать ее максимально простой для понимания.

Разбор выражений: постановка проблемы

Если вы никогда не задумывались о разборе выражений, вы вполне можете счесть его очень простой задачей. Однако, для того, чтобы более четко понять задачи, стоящие перед программистом, взявшимся за задачу по разбору выражений, попробуйте выполнить оценку следующего выражения:

```
10 - 2*3
```

Вы с легкостью определите, что значение этого выражения равно 4. Вы без труда можете создать программу, которая будет вычислять это конкретное выражение. Суть задачи, однако, заключается не в этом, а в том, чтобы создать программу, которая бы вычисляла любое произвольное выражение. Новичок, никогда ранее не занимавшийся этой проблемой, мог бы предложить что-нибудь вроде:

```
a = получить первый операнд
while(есть операнды) {
    op = получить оператор
    b = получить второй операнд
    a = a op b
}
```

Эта процедура получает первый операнд, затем оператор, затем — второй операнд, выполняет первую операцию, затем получает следующие оператор и операнд — если они имеются — и выполняет следующую операцию, и так далее. Однако, если вы попробуете воспользоваться этим подходом для вычисления выражения 10 - 2*3, то вместо правильного результата (4) вы получите 24 (8*3), так как эта процедура не учитывает приоритеты алгебраических операций. Если вы хотите получать правильные результаты, вы не можете последо-

вательно выполнять все операции слева направо, так как законы алгебры требуют, например, чтобы умножение выполнялось раньше вычитания. Новички считают, что эту проблему легко решить, и иногда — в очень ограниченных случаях — это действительно так. Однако, по мере того, как к разбираемым выражениям добавляются скобки, операторы возведения в степень, переменные и унарные операторы, проблема все усложняется и усложняется.

Существует несколько способов написания процедуры, разбирающей выражения. Пример, обсуждаемый в данной главе, является наиболее простым для реализации и одновременно — самым распространенным. Используемый для решения задачи метод называется *рекурсивно-нисходящим алгоритмом разбора выражений*. В процессе его подробного изучения вам несложно будет понять происхождение этого названия. Некоторые другие методы, используемые при написании программ разбора выражений, используют сложные таблицы, порождаемые другой программой. Программы, написанные с использованием этих методов, называются *программами разбора с табличным управлением* (table-driven parsers).

Разбор выражения

Существует несколько способов разбора и вычисления выражения. Для того, чтобы разработать рекурсивно-нисходящий алгоритм, будем рассматривать выражения как *рекурсивные структуры данных*, а именно — *структуры*, в определении которых встречаются ссылки на самих себя. Например, если на данный момент ограничиться только скобками и четырьмя действиями арифметики, то все выражения могут быть определены с использованием следующих правил:

выражение (член [+член][-член]

член (коэффициент [*коэффициент][/коэффициент]

коэффициент (переменная, число или (выражение)

Квадратные скобки здесь обозначают необязательный элемент, а символ (означает "производится" (создается, порождается). Фактически, эти правила называются правилами порождения выражений (*production rules*). Таким образом, следуя этим правилам, вы можете дать следующее определение понятию "член": "Член производится произведением или частным коэффициентов". Обратите внимание, что при определении выражений порядок следования операторов является неявным,

Например, выражение

10 + 5*B

состоит из двух членов: 10 и 5*B. Второй член состоит из двух коэффициентов: 5 и B. Эти коэффициенты включают одно число и одну переменную.

С другой стороны, выражение

14*(7-C)

состоит из двух коэффициентов: 14 и (7 — С). Первый коэффициент представляет собой число, а второй — заключенное в скобки выражение. Это выражение, в свою очередь, состоит из двух членов, один из которых является числом, а второй представляет собой переменную.

Рекурсивно-нисходящий алгоритм разбора выражений представляет собой набор взаимно-рекурсивных функций, реализующих правила порождения выражений, работая "по цепочке". На каждом шаге программа разбора выражений выполняет указанные операции в алгебраически правильной последовательности. Для того, чтобы понять принципы использования правил порождения при разборе выражений, рассмотрим следующий пример:

9/3 – (100 +56)

Ниже приведена последовательность разбора этого выражения.

1. Получить первый член, 9/3.
2. Получить каждый коэффициент, порождающий этот член, и выполнить деление целых. Результатом будет 3.
3. Получить второй член, (100 + 56). На данном этапе начинается рекурсивный анализ второго подвыражения.
4. Получить значение каждого члена и выполнить их сложение. Результат будет равен 156.
5. Вернуться из рекурсивного вызова. Вычесть 156 из 3. Результат будет равен −153.

Не расстраивайтесь, если на данном этапе вы что-то недопонимаете или даже совсем запутались. Обсуждаемая концепция достаточно сложна, и уж во всяком случае к ней требуется привыкнуть. При рекурсивном анализе выражений следует иметь в виду следующие два основных фактора. Во-первых, правила приоритетов операторов выражены неявно при формулировке правил порождения. Во-вторых, обсуждаемый метод разбора и вычисления выражений очень близок к естественному методу, которым любой человек стал бы вычислять математическое выражение.

Оставшаяся часть данной главы посвящена разработке трех программ разбора выражений. Первая версия программы осуществляет разбор и вычисления выражений с плавающей точкой, состоящих только из постоянных значений. Далее возможности этой программы будут расширены путем обеспечения поддержки переменных. Наконец, в третьей версии представлен параметризованный класс, который может использоваться для разбора выражений любого типа.

Класс Parser

Программа разбора выражений построена на базе класса **parser**. На протяжении этой главы мы будем использовать три версии класса **parser**. Первая версия программы будет использовать нижеприведенный класс. Все последующие версии класса **parser** построены на базе нижеприведенной.

```
class parser {
    char *exp_ptr; // указатель на выражение
    char token[80]; // текущий элемент
    char tok_type; // тип элемента

    void eval_exp2(double &result);
    void eval_exp3(double &result);
    void eval_exp4(double &result);
    void eval_exp5(double &result);
    void eval_exp6(double &result);
    void atom(double &result);
    void get_token();
    void serror(int error);
    int isdelim(char c);
public:
    parser();
    double eval_exp(char *exp)
};
```

Класс **parser** содержит три закрытых переменных-члена. Выражение, значение которого должно быть вычислено, содержится в заканчивающейся нулем строке, указателем на которую является **exp_ptr**. Таким образом, наша программа будет разбирать и вычислять выражения, содержащиеся в стандартных ASCII-строках. Например, нижеприведенные примеры содержат строки, которые эта программа сможет вычислить.

"10 - 5"

"2*3.3/(3.1416*3.3)"

Когда программа начинает работу, указатель **exp_ptr** должен указывать на первый символ строки, содержащей выражение. В процессе выполнения программа должна последовательно обрабатывать строку до тех пор, пока не встретится символ ее завершения (NULL).

Значения двух других переменных-членов, **token** и **token_type**, подробно рассматриваются в последующих разделах данной главы.

Точкой входа для программы является функция **eval_exp( )**, которую следует вызывать, задавая в качестве аргумента указатель на анализируемое выражение. Функции **eval_exp2( )** ... **eval_exp6( )** вместе с функцией **atom( )** формиру-

ют рекурсивно-нисходящий алгоритм. Они реализуют расширенный набор правил порождения выражений, обсуждавшихся ранее. Во все последующие версии программы будет включена также функция **eval_exp1( )**.

Функция **serror( )** осуществляет обработку синтаксических ошибок в выражении. Функции **get_token( )** и **isdelim( )** используются для разбиения выражения на составные части. Эта процедура подробно рассматривается в следующем разделе.

Разбиение выражений

Для того, чтобы выполнить оценку выражений, необходимо уметь разбивать их на составные части. Поскольку эта операция является фундаментальной для разбора выражений, ее необходимо подробно изучить прежде, чем вы сможете продвигаться далее.

Каждый компонент, формирующий выражение, называется элементом (*token*). Например, выражение

A*B -(W + 10)

содержит следующие элементы: A, *, B, -, (, W, +, 10,). Каждый элемент представляет собой неделимую часть выражения. В общем случае для реализации программы разбора выражений необходима функция, которая последовательно, один за другим, возвращает каждый элемент выражения по отдельности. Помимо этого, функция должна пропускать пробелы и табуляторы, а также определять конец выражения. Для этой цели используется функция **get_token( )**, являющаяся функцией-членом класса **parser**.

Для разбора выражения мало иметь функцию, которая возвращает элементы выражения, требуется еще определять тип возвращаемого элемента. Для программы разбора, обсуждаемой в этой главе, нужны только три типа элементов: **VARIABLE**, **NUMBER** и **DELIMITER**. (Тип **DELIMITER** используется как для операторов, так и для скобок).

Функция **get_token( )** приведена ниже. Она получает следующий элемент из разбираемого выражения, на которое указывает **exp_ptr**, и помещает его в переменную-член **token**. Тип элемента она помещает в переменную-член **tok_type**.

```
// Получение следующего элемента
void parser::get_token()
{
    register char *temp;

    tok_type = 0;
    temp = token;
```

```
    *temp = '\0';

    if(!*exp_ptr) return; // конец выражения

    while(isspace(*exp_ptr)) ++exp_ptr; // пропускаем пробелы

    if(strchr("+-*/%^=()", *exp_ptr)) {
        tok_type = DELIMITER;
        //Продвигаемся к следующему символу
        *temp++ = *exp_ptr++;
    }
    else if(isalpha(*exp_ptr)) {
        while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
        tok_type = VARIABLE;
    }
    else If(isdigit(*exp_ptr)) {
        while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
        tok_type = NUMBER;
    }

    *temp = '\0';
}
//Если с представляет собой разделитель, возвращаем true
int parser::isdelim(char c)
{
    if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
        return 1;
    return 0;
}
```

Внимательно рассмотрим вышеприведенные функции. Проведя первичные инициализации, функция **get_token( )** выполняет первую проверку - она проверяет, не встретился ли нуль, завершающий выражение. Для этого она проверяет символ, на который указывает **exp_ptr.** Поскольку **exp_ptr** представляет собой указатель на анализируемое выражение, то если он указывает на нуль, это означает, что достигнут конец выражения. Если это не так, то выражение содержит элементы, которые можно извлечь. Если строка содержит начальные пробелы, **get_token( )** пропускает их. После этого **exp_ptr** будет указывать на число, переменную, оператор, или — если выражение завершается пробелами — нуль. Если следующий символ представляет собой оператор, он возвращается как строка в **token,** а **tok_type** присваивается значение **DELIMITER**. Если следующий элемент представляет собой алфавитный символ, он считается одной из переменных. Он помещается в **token**, а переменной **tok_type** присваивается значение **VARIABLE**. Если следующий символ представляет собой цифру, то будет прочитано все число, которое затем в виде строки будет помещено в **token**, а переменной **tok_type** будет присвоено значение **NUM-**

BER. Наконец, если следующий символ не принадлежит ни к одному из вышеперечисленных видов, программа считает, что достигнут конец выражения. В этом случае переменной **token** присваивается значение NULL, что сигнализирует о том, что достигнут конец выражения.

Как мы уже упоминали ранее, в целях облегчения понимания кода этой программы мы опустили значительную часть кода обработки ошибок, а также сделали ряд предположений. Например, любой нераспознанный символ эта программа оценивает как конец выражения. Кроме того, в этой версии переменные могут иметь любую длину, однако, значимой будет только первая буква. По мере расширения функциональных возможностей вашего приложения вы можете усложнять код программы, добавляя разнообразные функции по обработке ошибок.

Для того, чтобы лучше понять процесс разбиения выражения на элементы, изучим возвращаемые значения на каждом шаге работы программы. В качестве примера используем следующее выражение:

A + 100 – (B*C)/2

Элемент	Тип элемента
A	VARIABLE
+	DELIMITER
100	NUMBER
–	DELIMITER
(	DELIMITER
B	VARIABLE
*	DELIMITER
C	VARIABLE
)	DELIMITER
/	DELIMITER
2	NUMBER
null	Null

Имейте в виду, что **token** всегда содержит строку, завершающуюся нулем, даже в том случае, когда она состоит из единственного символа.

Простая программа разбора выражений

Ниже приведена первая версия программы разбора выражений. Она может осуществлять разбор выражений, которые состоят только из констант, операторов и скобок. Разбирать выражения, которые содержат переменные, эта версия не может.

```
/*  Этот модуль представляет собой рекурсивно-нисходящий
    алгоритм, не распознающий переменных.
*/

#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
enum types {DELIMITER = 1, VARIABLE, NUMBER};

class parser {
    char *exp_ptr; // указатель на выражение
    char token[80]; // текущий элемент
    char tok_type; // тип элемента

    void eval_exp2(double &result);
    void eval_exp3(double &result);
    void eval_exp4(double &result);
    void eval_exp5(double &result);
    void eval_exp6(double &result);
    void atom(double &result);
    void get_token();
    void serror(int error);
    int isdelim(char c);
public:
    parser();
    double eval_exp(char *exp)
};
//конструктор
parser::parser()
{
    exp_ptr = NULL;
}

// Точка входа
double parser::eval_exp(char *exp)
{
    double result;
    exp_ptr = exp;

    get_token();
    if(!*token) {
        serror(2); // нет выражения
        return 0.0;
```

```
    }
    eval_exp2(result);
    if(*token) serror(0); // последний элемент должен быть нулем
    return result;
}

// Сложение и вычитание двух членов
void parser::eval_exp2(double &result)
{
    register char op;
    double temp;

    eval_exp3(result);
    while((op = *token) == '+' || op == '-') {
        get_token();
        eval_exp3(temp);
            switch(op) {
                case '-':
                    result = result - temp;
                    break;
                case '+':
                    result = result + temp;
                    break;
                }
            }
}

// Умножение и деление
void parser::eval_exp3(double &result)
{
    register char op;
    double temp;

    eval_exp4(result);
    while((op = *token) == '*' || op == '/' || op == '%') {
        get_token();
        eval_exp4(temp);
        switch(op) {
            case '*':
                result = result*temp;
                break;
            case '/':
                result = result/temp;
                break;
```

```
                case '%':
                    result = (int) result % (int) temp;
                    break;
                }
            }
}
// возведение в степень
void parser::eval_exp4(double &result)
{
    double temp, ex;
    register int i;

    eval_exp5(result);
    if(*token=='^') {
        get_token();
        eval_exp4(temp);
        ex = result;
        if(temp==0.0) {
            result = 1.0;
            return;
        }
        for(t=(int)temp-1; t>0; —t) result =
result*(double)ex;
    }
}
// Унарный + или -
void parser::eval_exp5(double &result)
{
    register char op;

    op = 0;
    if(tok_type == DELIMITER) && *token== '+' || *token == '-')
    {
        op = *token;
        get_token();
    }
    eval_exp6(result);
    if(op == '-') result = -result;
}

// Обработка выражения в скобках
void parser:: eval_exp6(double &result)
    {
    if((*token == ')')) {
        get_token();
        eval_exp2(result);
```

```
        if(*token != ')')
            serror(1);
        get_token();
    }
    else atom(result);
}

// Получение значения числа
void parser::atom(double &result)
{
    switch(tok_type) {
        case NUMBER:
            result = atof(token);
            get_token();
            return;
        default:
            serror(0);
    }
}

// Отображение синтаксической ошибки
void parser::serror(int error)
{
    static char *e[]= {
        "Синтаксическая ошибка",
        "Незакрытые скобки"
        "Нет выражения для разбора"
    };
    cout << e[error] << endl;
}
// Получение следующего элемента
void parser::get_token()
{
    register char *temp;

    tok_type = 0;
    temp = token;
    *temp = '\0';

    if(!*exp_ptr) return;

    while(isspace(*exp_ptr)) ++exp_ptr;
    if(strchr("+-*/%^=()", *exp_ptr)) {
        tok_type = DELIMITER;
        *temp++ = *exp_ptr++;
```

```
    }
    else if(isalpha(*exp_ptr)) {
        while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
        tok_type = VARIABLE;
    }
    else If(isdigit(*exp_ptr)) {
        while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
        tok_type = NUMBER;
    }
    *temp = '\0';
}
//Если с представляет собой разделитель, возвращаем true
int parser::isdelim(char c)
{
    if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
        return 1;
    return 0;
}
```

Эта программа разбора выражений, в том виде, как она здесь представлена, может обрабатывать следующие операторы: +, –, *, /, %. Кроме того, она может выполнять целое возведение в степень (^) и операцию унарного минуса. Наконец, программа корректно обрабатывает скобки. Фактическая оценка выражения происходит при взаимно рекурсивных вызовах функций **eval_exp2( )** ... **eval_exp6( )** и функции **atom( )**, возвращающей значение числа. Приведенные в начале каждой функции комментарии описывают ее роль в разборе выражения.

Использование этой программы может продемонстрировать простая функция **main( )**.

```
main()
{
    char expstr[80];

    cout << "Для останова введите точку.\n";

    parser ob;

    for(;;) {
        cout << "Введите выражение: ";
        cin.getline(expstr, 79);
        if(*expstr=='.') break;
        cout << "Ответ: " << ob.eval_exp(expstr) << "\n\n";
        };
        return 0;
}
```

Ниже приведен пример вывода этой программы:

Для останова введите точку.

Введите выражение: 10-2*3

Ответ: 4

Введите выражение: (10-2)*3

Ответ: 24

Введите выражение: 10/3

Ответ: 3.33333

Введите выражение: .

Принципы работы программы разбора выражений

Принципы работы программы разбора выражений исследуем на примере следующего выражения. (При этом будем подразумевать, что указатель **exp_ptr** указывает на начало выражения.)

10-3*2

При вызове функции **eval_exp( )**, являющейся точкой входа в программу разбора, она выделяет первый элемент выражения. Если этот элемент представляет собой NULL, функция выводит сообщение "Нет выражения для разбора" и возвращает управление. В нашем случае, однако, первый элемент содержит число **10**. Поскольку первый элемент - ненулевой, будет вызвана функция **eval_exp2( )**. После этого функция **eval_exp2( )** вызывает **eval_exp3( )**, **eval_exp3()** вызывает **eval_exp4( )**, а эта функция, в свою очередь - **eval_exp5( )**. Эта функция проверит, не является ли элемент унарным плюсом или минусом, и, поскольку в данном случае это не так, будет вызвана функция **eval_exp6( )**. На данном этапе функция **eval_exp6( )** рекурсивно вызывает **eval_exp2( )** (в том случае, если встретилось выражение, заключенное в скобки) или функцию **atom()**, определяющую значение числа. Поскольку в рассматриваемом случае нам не встретилось выражение, заключенное в скобки, будет вызвана функция **atom()**, в результате выполнения которой переменной **result** будет присвоено значение 10. После этого будет получен следующий элемент, и вся цепочка будет пройдена заново. В данном случае элемент представляет собой оператор **–**, и вызовы функций дойдут **до eval_exp2( )**.

То, что произойдет далее, исключительно важно. Поскольку элемент представляет собой оператор –, он будет сохранен в переменной **op**. После этого программа получит следующий элемент, представляющий собой число **3**, и прохождение цепочки начнется вновь. Как и ранее, в результате серии вызовов будет достигнута функция **atom( )**. Она возвратит через **result** значение **3**, после чего будет прочитан следующий элемент — оператор *****. Это вызовет возврат по цепочке к **eval_exp3( )**, где будет прочитан последний элемент — **2**. На этой стадии будет выполнена первая арифметическая операция — умножение **2** и **3**. Результат будет возвращен функции **eval_exp2( )**, которая выполнит вычитание. Операция вычитания даст ответ **4**. На первый взгляд этот процесс кажется сложным, поэтому рекомендуется проверить его на других примерах, чтобы твердо уяснить его механизм и убедиться в том, что он в любом случае дает правильный результат.

Вышеприведенную версию программы можно использовать при разработке программ-калькуляторов. Однако, для того, чтобы программу можно было использовать в более сложных приложениях — таких, как языки программирования, базы данных, сложные калькуляторы, ее необходимо расширить возможностями работы с переменными. Эта тема обсуждается в следующем разделе.

Включение в программу разбора выражений возможность работы с переменными

Переменные, используемые для хранения значений, предназначенных для последующего использования, применяются во всех языках программирования, многих продвинутых программах калькуляторов и электронных таблиц. Поэтому прежде, чем нашу разработку можно будет использовать для написания такого рода приложений, ее необходимо усовершенствовать, включив возможности работы с переменными. Для этого в код программы необходимо внести дополнения. Как уже говорилось ранее, алфавитные символы от **A** до **Z** будут использоваться для хранения переменных. Переменные будут храниться в массиве внутри класса **parser**. Каждая переменная будет использовать только один из 26 элементов массива типа **double**. Таким образом, к классу **parser** необходимо добавить следующее:

```
double vars[NUMVARS]; // значения переменных
```

Помимо этого, потребуется внести изменения в конструктор **parser**:

```
// конструктор parser
parser::parser()
```

```
{
    int I;
    exp_ptr = NULL;

    for (i=0; i<NUMVARS; I++) vars[i] = 0.0;
}
```

Как видно из вышеприведенного примера, для удобства пользователя все переменные инициализируются нулем.

Наконец, для наблюдения за значениями переменных вам потребуется специальная функция. Поскольку для имен переменных используются алфавитные символы от **A** до **Z**, их удобно использовать для индексации массива **vars** путем вычитания ASCII-кода символа **A** из имени переменной. Эту задачу выполняет функция-член **find_var( )**, приведенная ниже:

```
// Возврат значения переменной
double parser::find_var(char *s)
{
    if(!isalpha(*s)) {
        serror(1);
        return 0.0;
    return vars[toupper(*token) - 'A'];
}
```

Эта функция, в том виде, как она написана, фактически будет воспринимать и длинные имена переменных, однако, в таких именах значимым будет только первый символ. При написании конкретных приложений вы можете изменить эту функцию в соответствии с вашими потребностями.

Кроме того, потребуется модификация функции **atom( )**, с тем чтобы она могла обрабатывать как числа, так и переменные. Ниже представлена ее новая версия:

```
// Получение значения числа или переменной
void parser::atom(double &result)
{
    switch(tok_type) {
        case VARIABLE:
            result = find_var(token);
            get_token();
            return;
        case NUMBER:
            result = atof(token);
            get_token();
```

```
            return;
        default:
            serror(0);
    }
}
```

Эти дополнения представляют собой все, что технически необходимо для того, чтобы программа разбора выражений корректно обрабатывала переменные; однако, каким образом этим переменным будут присваиваться значения? Присваивание часто выполняется вне программы разбора выражений, однако, для этой цели можно использовать оператор присваивания, являющийся частью программы разбора. Сделать это можно несколькими способами. Один из них заключается в добавлении к классу **parser** еще одной функции, **eval_exp1( )**. Как следует из ее имени, эта функция будет начинать рекурсивно-нисходящую цепочку. Теперь именно она, а не функция **eval_exp2( )**, будет вызываться из функции **eval_exp( )** и фактически начинать процедуру разбора выражения. Код функции **eval_exp1( )** приведен ниже.

```
// Обработка присваиваний
void parser::eval_exp1(double &result)
{
    int slot;
    char ttok_type;
    char temp_token[80];

    if(tok_type==VARIABLE) {
        // сохранить старый элемент
        strcpy(temp_token, token);
        ttok_type = tok_type;

        // вычисление индекса переменной
        slot = toupper(*token) - 'A';

        get_token();
        if(*token != '=') {
            putback();вернуть текущий элемент
            // восстанавливаем предыдущий элемент
            strcpy(token, temp_token);
            tok_type = ttok_type;
        }
        else {
            get_token(); // следующая часть выражения
            eval_exp2(result);
            vars[slot] = result;
            return;
        }
```

```
    }

    eval_exp2(result);
}
```

Как видно из вышеприведенной распечатки, эта функция должна заглядывать вперед для того, чтобы определить, на самом ли деле имеет место присваивание. Это происходит потому, что имя переменной всегда предшествует оператору присваивания, но само по себе наличие имени переменной еще не гарантирует, что операция присваивания действительно произойдет. Таким образом, наша программа сможет "понять", что выражение **A = 100** представляет собой присваивание, а такое выражение, как **A/10**, присваиванием не является. Для этого функция **eval_exp1( )** считывает следующий элемент из вводного потока. Если он не является знаком равенства, элемент с помощью функции **putback( )** возвращается в вводной поток для дальнейшего использования. Функцию **putback( )** также необходимо включить в класс **parser**. Листинг этой функции приведен ниже:

```
// Возврат элемента во вводной поток
void parser::putback()
{
    char *t;

    t = token;
    for(; *t; t++) exp_ptr --;
}
```

После внесения всех необходимых изменений новая версия программы будет выглядеть следующим образом:

```
/*  Этот модуль представляет собой рекурсивно-нисходящий
    алгоритм, распознающий переменные.
*/

#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

enum types {DELIMITER = 1, VARIABLE, NUMBER};

const int NUMVARS = 26;

class parser {
    char *exp_ptr; // указатель на выражение
    char token[80]; // текущий элемент
```

```
    char tok_type; // тип элемента
    double vars[NUMVARS]; // массив значений переменных

    void eval_exp1(double &result);
    void eval_exp2(double &result);
    void eval_exp3(double &result);
    void eval_exp4(double &result);
    void eval_exp5(double &result);
    void eval_exp6(double &result);
    void atom(double &result);
    void get_token();
    void putback();
    void serror(int error);
    double find_var(char *s);
    int isdelim(char c);
public:
    parser();
    double eval_exp(char *exp)
};
//конструктор
parser::parser()
{
    int I;

    exp_ptr = NULL;

    for(i=0; i<NUMVARS; i++) vars[i] = 0.0;
}

// Точка входа
double parser::eval_exp(char *exp)
{
    double result;
    exp_ptr = exp;

    get_token();
    if(!*token) {
        serror(2); // нет выражения
        return 0.0;
    }
    eval_exp1(result);
    if(*token) serror(0); // последний элемент должен быть нулем
    return result;
}

// Обработка присваиваний
void parser::eval_exp1(double &result)
```

```
{
    int slot;
    char ttok_type;
    char temp_token[80];

    if(tok_type==VARIABLE) {
        // сохранить старый элемент
        strcpy(temp_token, token);
        ttok_type = tok_type;

        // вычисление индекса переменной
        slot = toupper(*token) - 'A';

        get_token();
        if(*token != '=') {
            putback();вернуть текущий элемент
            // восстанавливаем предыдущий элемент
            strcpy(token, temp_token);
            tok_type = ttok_type;
        }
        else {
            get_token(); // следующая часть выражения
            eval_exp2(result);
            vars[slot] = result;
            return;
        }
    }

    eval_exp2(result);
}
// Сложение и вычитание двух членов
void parser::eval_exp2(double &result)
{
    register char op;
    double temp;

    eval_exp3(result);
    while((op = *token) == '+' || op == '-') {
        get_token();
        eval_exp3(temp);
            switch(op) {
                case '-':
                    result = result - temp;
                    break;
```

```
                    case '+':
                         result = result + temp;
                         break;
                    }
               }
}
// Умножение и деление
void parser::eval_exp3(double &result)
{
     register char op;
     double temp;

     eval_exp4(result);
     while((op = *token) == '*' || op == '/' || op == '%') {
          get_token();
          eval_exp4(temp);
          switch(op) {
               case '*':
                    result = result*temp;
                    break;
               case '/':
                    result = result/temp;
                    break;
               case '%':
                    result = (int) result % (int) temp;
                    break;
               }
          }
}
// Возведение в степень
void parser::eval_exp4(double &result)
{
     double temp, ex;
     register int i;

     eval_exp5(result);
     if(*token=='^') {
          get_token();
          eval_exp4(temp);
          ex = result;
          if(temp==0.0) {
               result = 1.0;
```

```
// Унарный + или -
void parser::eval_exp5(double &result)
{
     register char op;

     op = 0;
     if(tok_type == DELIMITER) && *token== '+' || *token == '-')
     {
          op = *token;
          get_token();
     }
     eval_exp6(result);
     if(op == '-') result = -result;
}

// Обработка выражения в скобках
void parser:: eval_exp6(double &result)
     if((*token == ')')) {
          get_token();
          eval_exp2(result);
          if(*token != ')')
               serror(1);
          get_token();
     }
     else atom(result);
}

// Получение значения числа или переменной
void parser::atom(double &result)
{
     switch(tok_type) {
          case VARIABLE:
               result = find_var(token);
               get_token();
               return;
          case NUMBER:
               result = atof(token);
               get_token();
               return;
          default:
               serror(0);
     }
}
// Возврат элемента во вводной поток
void parser::putback()
```

```
{
     char *t;
     t = token;
     for(; *t; t++) exp_ptr --;
}
// Отображение синтаксической ошибки
void parser::serror(int error)
{
     static char *e[]= {
          "Синтаксическая ошибка",
          "Незакрытые скобки"
          "Нет выражения для разбора"
     }
     cout << e[error] << endl;
}
// Получение следующего элемента
void parser::get_token()
{
     register char *temp;

     tok_type = 0;
     temp = token;
     *temp = '\0';

     if(!*exp_ptr) return;

     while(isspace(*exp_ptr)) ++exp_ptr;
     if(strchr("+-*/%^=()", *exp_ptr)) {
          tok_type = DELIMITER;
          *temp++ = *exp_ptr++;
     }
     else if(isalpha(*exp_ptr)) {
          while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
          tok_type = VARIABLE;
     }
     else If(isdigit(*exp_ptr)) {
          while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
          tok_type = NUMBER;
     }

     *temp = '\0';
}
//Если с представляет собой разделитель, возвращаем true
int parser::isdelim(char c)
```

```
{
    if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
        return 1;
    return 0;
}
// Возврат значения переменной
double parser::find_var(char *s)
{
    if(!isalpha(*s)) {
        serror(1);
        return 0.0;
    return vars[toupper(*token) - 'A'];
}
```

Для проверки работы усовершенствованной программы можно использовать ту же самую функцию **main()**, которая использовалась для проверки работоспособности первой версии. Новая версия программы позволит вводить выражения типа:

A=10/4

A - B

C = A*(F-21)

Синтаксическая проверка в рекурсивно-нисходящем алгоритме разбора выражений

Прежде, чем переходить к параметризованной версии нашей программы разбора выражений, необходимо кратко рассмотреть вопросы синтаксической проверки. Синтаксическая ошибка в процессе разбора выражения представляет собой ситуацию, при которой введенное для разбора выражение не соответствует жестким условиям, предъявляемым программой разбора. В большинстве случаев синтаксические ошибки являются следствием ошибок человека — например, опечаток при вводе выражения. Например, исследуемые в данной главе программы воспримут нижеприведенные выражения как ошибочные:

10 ** 8

(10–5)*9)

/8

Первое выражение содержит два смежных оператора, во втором не совпадает количество открывающих и закрывающих скобок, а третье содержит знак деления в самом начале выражения. Ни одно из этих условий не допускается нашей программой. Поскольку синтаксические ошибки в лучшем случае приведут к неверному результату, необходимо предусмотреть в программе их обработку.

Исследуя код программы, вы наверняка обратили внимание на функцию **serror( )**, вызываемую при определенных обстоятельствах. В отличие от многих других рекурсивно-нисходящий алгоритм значительно упрощает синтаксическую проверку, которая по большей части имеет место в функциях **atom( )**, **find_var( )** и **eval_exp6( )**, где производится подсчет количества скобок. Единственной проблемой, существующей при обработке синтаксических ошибок, в том виде, как она на данный момент реализована, является то, что когда встречается синтаксическая ошибка, выполнение программы не прерывается. Неприятным результатом этого может явиться вывод большого количества раздражающе однообразных сообщений об ошибках.

Наилучшим вариантом, позволяющим избежать этого, является такая реализация функции **serror()**, при которой она выполняет тот или иной вид сброса или переинициализации. Так, например, все компиляторы C++ содержат пару взаимодополняющих функций **setjmp()** и **longjmp()**. Эти функции позволяют осуществлять ветвление программы с переходом к другой функции. Таким образом, в случае синтаксической ошибки функция **serror( )** могла бы выполнять функцию **longjmp( )** и переходить в безопасную точку, расположенную за пределами модуля разбора выражений.

Кроме того, в зависимости от того, какое применение вы найдете программе разбора выражений, может оказаться очень полезным механизм обработки исключений C++ (реализованный через **try**, **catch** и **throw**).

Если вы оставите код программы без изменений, при синтаксических ошибках она будет выводить множество сообщений об ошибках. Впрочем, если в некоторых случаях это не приносит ничего, кроме раздражения, то в других, наоборот, может сослужить добрую службу, позволив выявить множество ошибок. Разумеется, если вы хотите использовать программу в коммерческих продуктах, синтаксическую проверку следует усовершенствовать.

Построение параметризованной версии программы разбора выражений

Обе предыдущих версии работали с числовыми выражениями, для которых подразумевалось, что все значения в них имеют тип **double**. Разумеется, для приложений, использующих значения типа **double**, это очень даже хорошо. Но если ваше приложение работает только с целыми числами, то это будет и или

шеством. Кроме того, жесткое декларирование типа значений в разбираемых выражений накладывает на программу множество ненужных ограничений. К счастью, несложно построить параметризованный класс **parser**, который сможет обрабатывать данные любых типов, для которых определены алгебраические выражения. Как только это будет сделано, программу можно будет использовать с любыми встроенными типами данных, а также с любыми числовыми типами, которые вы можете самостоятельно определить.

Ниже приведена параметризованная версия программы разбора выражений:

```
// Параметризованная программа разбора выражений

#include <iostream.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

enum types {DELIMITER = 1, VARIABLE, NUMBER};

const int NUMVARS = 26;

template <class PType> class parser {
     char *exp_ptr; // указатель на выражение
     char token[80]; // текущий элемент
     char tok_type; // тип элемента
     PType vars[NUMVARS]; // массив значений переменных

     void eval_exp1(double &result);
     void eval_exp2(double &result);
     void eval_exp3(double &result);
     void eval_exp4(double &result);
     void eval_exp5(double &result);
     void eval_exp6(double &result);
     void atom(PType &result);
     void get_token();
     void putback();
     void serror(int error);
     PType find_var(char *s);
     int isdelim(char c);
public:
     parser();
     PType eval_exp(char *exp)
};
//конструктор
template <class Ptype> parser<PType>::parser()
```

```
{
    int i;

    exp_ptr = NULL;
    for(i=0; i<NUMVARS; i++) vars[i] = (PType) 0;
}

// Точка входа
template <class PType> Ptype parser<PType>::eval_exp(char *exp)
{
    PType result;
    exp_ptr = exp;

    get_token();
    if(!*token) {
        serror(2); // нет выражения
        return (Ptype) 0;
    }
    eval_exp1(result);
    if(*token) serror(0);//последний элемент должен быть нулем
    return result;
}

// Обработка присваиваний
template<class Ptype> void parser<PType>::eval_exp1(PType &result)
{
    int slot;
    char ttok_type;
    char temp_token[80];

    if(tok_type==VARIABLE) {
        // сохранить старый элемент
        strcpy(temp_token, token);
        ttok_type = tok_type;

        // вычисление индекса переменной
        slot = toupper(*token) - 'A';

        get_token();
        if(*token != '=') {
            putback();вернуть текущий элемент
            // восстанавливаем предыдущий элемент
            strcpy(token, temp_token);
            tok_type = ttok_type;
        }
```

```
                get_token(); // следующая часть выражения.
                eval_exp2(result);
                vars[slot] = result;
                return;
            }
        }

        eval_exp2(result);
}
// Сложение и вычитание двух членов
template <class Ptype> void parser<PType>::eval_exp2(PType &result)
{
        register char op;
        PType temp;

        eval_exp3(result);
        while((op = *token) == '+' || op == '-') {
            get_token();
            eval_exp3(temp);
                switch(op) {
                    case '-':
                        result = result - temp;
                        break;
                    case '+':
                        result = result + temp;
                        break;
                    }
                }
}
// Умножение и деление
template <class Ptype> void parser<PType>::eval_exp3(PType &result)
{
        register char op;
        PType temp;

        eval_exp4(result);
        while((op = *token) == '*' || op == '/' || op == '%') {
            get_token();
            eval_exp4(temp);
            switch(op) {
                case '*':
                    result = result*temp;
                    break;
                case '/':
```

```
                result = result/temp;
                break;
            case '%':
                result = (int) result % (int) temp;
                break;
            }
        }
}
// Возведение в степень
template <class Ptype> void parser<PType>::eval_exp4(PType &result)
{
    PType temp, ex;
    register int i;

    eval_exp5(result);
    if(*token=='^') {
        get_token();
        eval_exp4(temp);
        ex = result;
        if(temp==0.0) {
            result = (Ptype) 1;
            return;
        }
        for(t=(int)temp-1; t>0; —t) result =
result*(double)ex;
    )
}
// Унарный + или -
template <class Ptype> void parser<PType>::eval_exp5(PType &result)
{
    register char op;

    op = 0;
    if(tok_type == DELIMITER) && *token== '+' || *token == '-')
    {
        op = *token;
        get_token();
    }
    eval_exp6(result);
    if(op == '-') result = -result;
}
// Обработка выражения в скобках
template <class Ptype> void parser<PType>:: eval_exp6(PType &result)
    {
    if((*token == '(')) {
        get_token();
```

```
            eval_exp2(result);
            if(*token != ')')
                 serror(1);
            get_token();
      }
      else atom(result);
}

// Получение значения числа или переменной
template <class Ptype> void parser<PType>::atom(PType &result)
{
      switch(tok_type) {
            case VARIABLE:
                 result = find_var(token);
                 get_token();
                 return;
            case NUMBER:
                 result = (Ptype) atof(token);
                 get_token();
                 return;
            default:
                 serror(0);
      }
}
// Возврат элемента во вводной поток
template <class Ptype> void parser<PType>::putback()
{
      char *t;

      t = token;
      for(; *t; t++) exp_ptr —;
}
// Отображение синтаксической ошибки
template <class Ptype> void parser<PType>::serror(int error)
{
      static char *e[]= {
            "Синтаксическая ошибка",
            "Незакрытые скобки"
            "Нет выражения для разбора"
      };
      cout << e[error] << endl;
}
// Получение следующего элемента
template <class Ptype> void parser<PType>::get_token()
```

```
{
    register char *temp;
    tok_type = 0;
    temp = token;
    *temp = '\0';

    if(!*exp_ptr) return;

    while(isspace(*exp_ptr)) ++exp_ptr;
    if(strchr("+-*/%^=()", *exp_ptr)) {
        tok_type = DELIMITER;
        *temp++ = *exp_ptr++;
    }
    else if(isalpha(*exp_ptr)) {
        while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
        tok_type = VARIABLE;
    }
    else If(isdigit(*exp_ptr)) {
        while(!isdelim(*exp_ptr)) *temp++ = *exp_ptr++;
        tok_type = NUMBER;
    }

    *temp = '\0';
}
//Если с представляет собой разделитель, возвращаем true
template <class Ptype> int parser<PType>::isdelim(char c)
{
    if(strchr(" +-/*%^=()", c) || c==9 || c=='\r' || c==0)
        return 1;
    return 0;
}
// Возврат значения переменной
template <class Ptype> PType parser<PType>::find_var(char *s)
{
    if(!isalpha(*s)) {
        serror(1);
        return (Ptype) 0;
    return vars[toupper(*token) - 'A'];
}
```

Как видите, теперь тип данных, над которыми оперирует алгоритм разбора выражений, указан параметризованным типом Ptype. Проверить работу параметризованной версии можно с помощью следующей функции main():

```
main()
{
     char expstr[80];
// Демонстрация разбора выражений с плавающей точкой
     parser <double> ob;

     cout << "Разбор выражений с плавающей точкой. ";
     cout << "Для останова введите точку\n";
     for(;;) {
          cout << "Введите выражение: ";
          cin. getline(expstr, 79);
          if(*expstr=='.') break;
          cout << "Ответ: " << ob.eval_exp(expstr) << "\n\n";
     }
     cout << endl;
// Демонстрация разбора целых выражений
     parser<int> Iob;

     cout << "Разбор целых выражений. ";
     cout << "Для останова введите точку\n";
     for(;;) {
          cout << "Введите выражение: ";
          cin. getline(expstr, 79);
          if(*expstr=='.') break;
          cout << "Ответ: " << Iob.eval_exp(expstr) << "\n\n";
     }
     return 0;
}
```

Ниже приведен пример вывода данной программы:

Разбор выражений с плавающей точкой. Для останова введите точку

Введите выражение: a=10.1
Ответ: 10.1

Введите выражение: b=3.2
Ответ: 3.2

Введите выражение: a/b
Ответ: 3.15625

Введите выражение:.

```
Разбор целых выражений. Для останова введите точку
Введите выражение: a=10
Ответ: 10

Введите выражение: b=3
Ответ:3

Введите выражение: a/b
Ответ: 3

Введите выражение:.
```

Как видите, при разборе выражений с плавающей точкой используются числа с плавающей точкой, а при разборе целых выражений используются целые значения.

Рекомендации для самостоятельной разработки

Как уже говорилось ранее в этой главе, представленная здесь версия программы разбора выражений реализует только минимально необходимую обработку ошибок. Возможно, вам требуется детальная диагностика ошибок. Например, можно выделять ту позицию в выражении, в которой найдена ошибка. Это позволит пользователю найти и исправить синтаксическую ошибку.

Программа разбора выражений, в том виде, как она здесь представлена, может выполнять разбор только числовых выражений. Ее возможности можно расширить, дополнив ее обработкой таких типов данных, как строки, пространственные координаты и комплексные числа. Например, для того, чтобы представленная здесь программа могла выполнять разбор строк, в ее код необходимо внести следующие изменения:

1. Определить класс строк, включающий все необходимые операторы.
2. Определить новый тип элемента с именем TOKEN.
3. Модифицировать функцию **get_token( )** таким образом, чтобы она могла распознавать строки, заключенные в двойные кавычки.
4. Добавить в состав функции **atom( )** вариант, обрабатывающий элементы типа TOKEN.

После того, как вы правильно реализуете эти модификации, программа сможет выполнять над строками операции следующего типа:

```
a="one"
```

b="two"

c=a+b

Результат выполнения этой операции, помещаемый в переменную **c**, должен представлять собой конкатенацию строк **a** и **b**, или "**onetwo**"

Создайте простое приложение, которое использует всплывающее окно, считывает введенное пользователем выражение, а затем отображает результат выполнения операции. Эта разработка пригодится вам практически для любого коммерческого приложения. Если вы программируете для Windows, сделать это будет особенно просто.

Глава 4

4

Разреженные массивы в стиле С++

Одной из наиболее интригующих проблем программирования является реализация *разреженного массива*. Разреженным называется массив, в котором не все элементы фактически используются, присутствуют на своих местах или нужны. Разреженные массивы являются исключительно ценным средством в том случае, когда соблюдены оба из нижеприведенных условий: размер массива, необходимый для работы приложения, достаточно велик (возможно, даже превышает размер доступной памяти), и при этом далеко не все элементы массива будут использоваться. Таким образом, разреженный массив, как правило, будет массивом большого размера с малым количеством фактически присутствующих элементов. Из дальнейшего изложения будет видно, что разреженные массивы можно реализовать несколькими различными способами. Хотя методы реализации разреженных массивов, изложенные в этой главе, важны сами по себе, они преследуют и еще одну цель, а именно: на реальном примере показать, как наиболее сложные особенности языка С++ могут быть использованы для получения простых и элегантных решений классических проблем программирования. Кроме того, поскольку С++ обладает возможностью перегрузки оператора **[ ]**, разреженные массивы можно полностью интегрировать в вашу среду программирования.

В этой главе рассматриваются четыре метода построения параметризованного разреженного массива: связный список, бинарное дерево, массив указателей и хэширование (**hashing**). Каждый из подходов реализован как параметризованный класс. Таким образом, любой из обсуждаемых здесь методов можно использовать для построения разреженных массивов, состоящих из элементов, принадлежащих к любому типу данных. Классы разреженных массивов построены путем модификации и расширения контейнерного класса ограниченного массива, который обсуждался в главе 2.

Однако, прежде, чем начинать обсуждение, рассмотрим проблему, для решения которой были разработаны разреженные массивы.

Цели разработки разреженных массивов

Для того, чтобы лучше понять, зачем же нужны разреженные массивы, рассмотрим следующие два положения:

- При декларировании обычного массива C++ происходит единовременное выделение всей памяти, необходимой под этот массив.
- Большие массивы — особенно многомерные — могут занимать очень большие объемы памяти.

Рассмотрим ограничения, вытекающие из этих положений. Тот факт, что память, необходимая для хранения массива, выделяется именно в момент его создания, означает, что предельный размер массива, который вы можете декларировать в своей программе, ограничивается (частично) объемом доступной памяти. Если для работы вашего приложения нужен массив, размер которого превышает физический объем памяти, имеющейся в вашем компьютере, то для поддержки массива вам придется использовать другой механизм. (Например, большой неразреженный массив как правило, использует ту или иную форму виртуальной памяти.) Даже в тех случаях, когда большой массив и может разместиться в памяти, идея поступить таким образом не слишком удачна, поскольку это может оказать отрицательное влияние как на работу вашей программы, так и на всю компьютерную систему в целом. Как уже говорилось, если вы, программируя на C++, декларируете массив, вся память, необходимая для хранения этого массива, выделяется единовременно. Это означает, что выделенная для хранения массива память будет недоступна другим частям вашей программы или процессам операционной системы. Таким образом, метод выделения памяти для хранения большого массива исключительно нерационально использует системные ресурсы, а в тех случаях, когда массив является разреженным (то есть не все его элементы присутствуют или фактически используются), это особенно справедливо.

Таким образом, массивы большой размерности с малым количеством элементов создают серьезные проблемы, для решения которых были разработаны разнообразные методы. Все методы работы с разреженными массивами имеют одну общую черту: память для хранения элементов массива выделяется только по мере надобности. Таким образом, все методы работы с разреженными массивами обладают тем преимуществом, что для их работы требуются меньшие объемы памяти, которая выделяется под хранение только тех элементов массива, которые фактически используются в работе программы. Око

помленная память используется для других целей. Кроме того, благодаря этим методам можно использовать очень крупные массивы, размеры которых превышают объемы, допускаемые системными ресурсами.

В качестве примеров можно привести множество приложений, для работы которых требуется обработка разреженных массивов. Многие из них связаны с анализом матриц и такими научными и инженерными задачами, которые понятны только экспертам в соответствующей области. Однако, существует и пример использующего разреженные массивы приложения, хорошо известного и понятного всем — это электронные таблицы. Несмотря на то, что матрица средней электронной таблицы может быть очень велика, в любой конкретный момент времени использоваться будет только некоторая ее часть. Электронные таблицы используют матрицу для хранения формул, значений и строк, ассоциируемых с каждой из ячеек. При использовании разреженных массивов память для хранения каждого из элементов выделяется из пула свободной памяти только по мере надобности. При этом, поскольку фактически используется только незначительная часть элементов массива, массив (электронная таблица) при своих больших размерах будет потреблять память для хранения только тех элементов, которые действительно используются.

На протяжении всей этой главы мы будем активно использовать следующие два термина: *логический массив* и *физический массив*. Логический массив является воображаемым, а физический массив — это массив, который фактически существует в системе. Например, если вы определите следующий разреженный массив:

```
ArrayType arrayob[100000];
```

то логический массив будет состоять из 100000 элементов даже в том случае, если этот массив в системе физически не существует. Таким образом, если фактически используются только 100 элементов этого массива, то только эти 100 элементов будут потреблять физическую память компьютера. Разработанные в этой главе методы работы с разреженными массивами устанавливают связь между физическими и логическими массивами.

Объекты типа разреженных массивов

Большинство реализаций методов работы с разреженными массивами, включая обсуждаемые в этой главе, используют тот или иной тип объектов для хранения фактически используемых элементов массива. Чаще всего этот объект будет содержать как минимум два информационных элемента: логический индекс элемента и его значение. Однако, в зависимости от используемого подхода может потребоваться больший или меньший объем информации. Классы разреженных массивов, представленные в этой главе, используют объекты типа **ArrayOb**. Ниже приведена версия этого класса, используемая тремя из четырех обсуждаемых здесь методов. (Четвертый метод работы с разреженными массивами использует незначительно измененную, но похожую версию.)

```
/*
Класс, определяющий тип объекта, хранящегося в разреженном массиве
*/
template <class DataT> class ArrayOb {
public:
     long index; // индекс элемента массива
     DataT info; // информация
}
```

Здесь **index** содержит логический индекс элемента а **info** — его значение. Например, представим себе разреженный массив **Sparse**. Тогда утверждение:

Sparse[987343] = 10

сохранит объект типа **ArrayOb** в физическом массиве, причем поле **index** будет содержать значение 987343, а поле **info** - значение 10.

Как видите, **ArrayOb** является параметризованным классом, в котором тип данных указан параметризованным типом **DataT**. Значение **index** имеет тип **long**, что обеспечивает поддержку массивов больших размерностей. Этот класс будет наследоваться классами разреженных массивов, обсуждаемыми на протяжении всей этой главы.

Разреженный массив на базе связного списка

Одним из наиболее простых методов реализации разреженных массивов является использование связного списка для хранения фактически используемых элементов массива. Таким образом, физический массив реализован в виде связного списка. Когда используется этот подход, при индексации логического массива выполняется поиск по списку, и возвращается значение, ассоциированное с индексом. Для того, чтобы класс связного списка, разработанный в главе 2, можно было использовать для поддержки метода реализации разреженных массивов, его необходимо переработать и дополнить.

Ниже приведен класс разреженного массива на основе связного списка, а также короткая функция **main( )**, демонстрирующая его использование.

```
/*
Параметризованный класс разреженного массива, реализованный
на основе связного списка
*/

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

```
/*
Этот класс определяет тип объекта, хранящегося в разреженном массиве
*/

template <class DataT> class ArrayOb {
public:
     long index; // индекс элемента массива
     DataT info; // информация
};

// Объект разреженного массива на базе связного списка

template <class DataT>
class SparseOb: public ArrayOb<DataT> {
public:
SparseOb<DataT> *next; // указатель на следующий объект
SparseOb<DataT> *prior; // указатель на предыдущий объект

SparseOb() {
     info = 0;
     index = -1;
     next = NULL;
     prior = NULL;
     };
};

/*
Параметризованный класс связного списка с двойными ссылками
для разреженных массивов
*/

template <class DataT>
class SparseList : public SparseOb<DataT> {
     SparseOb<DataT> *start, *end;
public:
     SparseList() {start = end = NULL;}
     ~SparseList();

     void store(long ix, DataT c); // сохранение элемента
     void remove(long ix); // удаление элемента

     // возвращение указателя на элемент по заданному индексу
     SparseOb<DataT> *find(long ix);
};
```

```
// Деструктор SparseList
template <class DataT> SparseList<DataT>:: ~SparseList()
{
     SparseOb<DataT> *p, *p1;
     // освобождение всех элементов списка
     p = start;
     while(p) {
          p1 = p->next;
          delete p;
          p = p1;
          }
     }

     // Добавление элемента в список
     template <class DataT>
     void SparseList<DataT>::store(long ix, DataT c)
     {
          SparseOb<DataT> *p;

          p = new SparseOb<DataT>;
          if(!p) {
               cout << "Ошибка выделения памяти.\n";
               exit(1);
               }
               p->info = c;
               p->index = ix;

               if(start == NULL) { // первый элемент списка
                    end = start = p;
               }
               else { // добавляем в конец списка
                    p->prior = end;
                    end->next = p;
                    end = p;
                    }
               }
/*
Удаление из массива элемента с указанным индексом и обновле-
ние указателей на начало и конец
*/

template <class DataT>
void SparseList<DataT>::remove(long ix)
{
SparseOb<DataT> *ob;
```

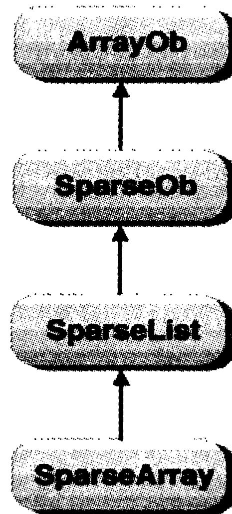
```
ob = find(ix); // получаем указатель на элемент
if(!ob) return; // элемент не существует

if(ob->prior) {// удаляется не первый элемент
     ob->prior->next = ob->next;
     if(ob->next) { // удаляется не последний элемент
          ob->next->prior = ob->prior;
     else // удаляется последний элемент
          end = ob->prior; // обновление указателя на конец
     }
     else { // Удаляется первый элемент
          if(ob->next) { // список не пуст
               ob->next->prior = NULL;
               start = ob->next;
               }
               else // теперь список пуст
                    start = end = NULL;
          }
}

// Нахождение элемента массива по индексу

template <class DataT>
SparseOb<DataT> *SparseList<DataT>::find(long ix)
{
     SparseOb<DataT> *temp;

     temp = start;

     while(temp) {
          if(ix==temp->index) return temp; // найдено вхожде-
ние
          temp = temp->next;
          }
          return NULL; // нет в списке
}

// Параметризованный класс разреженного массива

template <class DataT>
class SparseArray : public SparseList<DataT> {
     long length; // размерность массива
public:
     SparseArray(long size) { length = size; }
     DataT &operator[ ](long I);
     };
```

```
// Индексация в разреженном массиве
template <class DataT>
DataT &SparseArray<DataT>::operator[ ](long ix)
{
     if(ix<0 || ix>length-1) {
          cout << "\nЗначение индекса ";
          cout << ix << " выходит за пределы области определения.\n";
          exit(1);
     }
     SparseOb<DataT> *o;

     o = find(ix); // получаем указатель на элемент
     if(!o) { // новый элемент в массиве
          store(ix, 0); //сохраняем новый элемент
          o = find(ix); // получаем указатель на новый элемент
     }

     return o->info;
}

main()
{
SparseArray<int> iob(100000);
SparseArray<double> dob(100000);
int i;

cout << " Это массив целых: " << endl;

// Поместим в массив некоторые значения
for(i=0; i<5; i++) iob[i] = i;
for(i=0; i<5; i++) cout << iob[i] << " ";
cout << endl;

// Присвоим значение одного из элементов другому
iob[2] = iob[3];
for(i=0; i<5; i++) cout << iob[i] << " ";
cout << endl;

// Добавим еще несколько элементов
iob[1000] = 9345;
iob[2000] = iob[1000] + 100;

cout << iob[1000] << " " << iob[2000] << endl;

cout << " Это массив значений с плавающей точкой: " << endl;
for(i=0; i<5; i++) dob[i*100] = (double) i*1.19;
```

```
for(i=0; i<5; i++) cout << dob[1*100] << " ";
cout << endl;

// Попробуем получить доступ к позиции, не получившей значения
dob[200] = dob[999];
cout << dob[200] << " " << dob[999] << endl;

return 0;
}
```

Некоторые детали использования связного списка

Разреженный массив с использованием связного списка построен на базе многоуровневой иерархии параметризованных классов. На самом верхнем уровне этой иерархии находится класс **ArrayOb**. **ArrayOb** наследуется классом **SparseOb**, который, в свою очередь, наследуется классом **SparseList**, и, наконец, на последнем уровне иерархии классов находится класс **SparseArray**. Эту иерархию иллюстрирует нижеприведенная схема.



Классы связных списков **SparseOb** и **SparseList** представляют собой незначительно модифицированные версии классов **listob** и **dllist**, представленных в главе 2. Например, функция **find( )** теперь выполняет поиск индекса массива по связному списку. Однако, основополагающая логика этих классов не изменилась, и они не содержат ничего нового по сравнению с версиями из главы 2. (Если вы испытываете затруднения с пониманием принципов работы классов,

реализующих связные списки, перечитайте главу 2.) Хотя класс разреженного массива **SparseArray** не использует функцию-член **remove( )** класса **SparseList**, эта функция может оказаться очень полезной при дальнейшей разработке самостоятельных приложений. Например, ее можно использовать при выполнении очистки массива от элементов, ставших ненужными и помеченных для удаления.

Разреженные массивы декларируются путем конкретизации объектов типа **SparseArray** с указанием типа данных и размерности массива (как показано в примере). Поскольку связный список, используемый для поддержки разреженного массива, практически не ограничен, теоретически при создании разреженного массива нет и необходимости указывать фактический размер логического массива. Однако, благодаря указанию размерности логического массива, класс **SparseArray** получает возможность выявления индексов, выходящих за рамки границ массива.

Наиболее интересен следующий фрагмент кода класса **SparseArray** (для удобства восприятия мы приводим его еще раз):

```
// Параметризованный класс разреженного массива

template <class DataT>
class SparseArray : public SparseList<DataT> {
     long length; // размерность массива
public:
     SparseArray(long size) { length = size; }
     DataT &operator[ ](long I);
     };

// Индексация в разреженном массиве

template <class DataT>
DataT &SparseArray<DataT>::operator[ ](long ix)
{
     if(ix<0 || ix>length-1) {
          cout << "\nЗначение индекса ";
          cout << ix << " выходит за пределы области определения.\n";
          exit(1);
     }
     SparseOb<DataT> *o;

     o = find(ix); // получаем указатель на элемент
     if(!o) { // новый элемент в массиве
          store(ix, 0); //сохраняем новый элемент
          o = find(ix); // получаем указатель на новый элемент
     }
     return o->info;
}
```

Ключ к пониманию принципов разреженных массивов находится в функции **operator[ ]( )**, которая, как вы помните, инициируется каждый раз при индексации массива. В теле этой функции выполняется проверка диапазона индексов, с тем чтобы исключить ошибки, связанные с выходом за пределы границ массива. Затем функция пытается найти элемент в списке по заданному индексу. Если элемент уже находится в списке, функция возвращает ссылку на информацию, хранящуюся в нем. Если элемента в списке нет, он создается и помещается в список. Следующим шагом функция получает указатель на новый элемент и возвращает ссылку на содержащуюся в нем информацию. Этот простой, но эффективный механизм работает во всех случаях. В качестве примера рассмотрим следующий фрагмент кода:

```
SparseArray<int> ob(100000);
int x;

aob[1] = 10; // утверждение 1
x = ob[1]; утверждение 2
```

В первом утверждении при инициации функции **operator[ ]( )** ее параметр **ix** получает значение индекса, равное 1. Функция **operator[ ]( )** ищет этот индекс в списке. Так как это первая ссылка на индекс 1, в списке он найден не будет. Поэтому элемент с этим индексом будет добавлен в список и получит начальное значение, равное нулю. Затем будет получен указатель на этот объект. После этого функция возвратит ссылку на элемент **info** этого объекта. Эта ссылка будет использована для присвоения элементу **info**, ассоциированному с индексом 1, значения 10. Когда будет выполняться второе утверждение, функция **operator[ ]( )** снова будет выполняться со значением параметра **ix**, равным 1. На этот раз индекс 1 будет найден в списке, и будет возвращена ссылка на ассоциированный с ним член **info**. Поскольку в результате выполнения предыдущего утверждения член **info** получил значение 10, именно это значение и будет присвоено переменной **x**. Как вы можете видеть, память для хранения элементов массива выделяется только по мере их фактического использования.

Прежде, чем переходить к дальнейшему изучению, попробуйте поэкспериментировать с приведенным выше листингом программы, добавляя к нему разнообразные утверждения **cout**. Таким образом, вы сможете увидеть, что происходит при каждом вызове функции **operator[ ]( )**.

Анализ подхода с использованием связного списка

Принципиальное преимущества подхода к обработке разреженных массивов с использованием связных списков заключается в том, что он обеспечивает эффективное использование памяти — она выделяется для хранения только тех элементов массива, которые действительно содержат информацию. Кроме того, ме

тод прост в реализации. Несмотря на эти преимущества, метод имеет и существенный недостаток: он использует линейный поиск при доступе к элементам списка. Это означает, что время поиска будет пропорционально *n*/2, где *n* — количество элементов в списке. Этот недостаток не будет представлять собой проблемы при работе с сильно разреженными массивами. Однако, если физический массив содержит достаточно большое количество элементов, то время поиска станет неприемлемым. В этом случае вы сможете улучшить время поиска, использовав для поддержки разреженных массивов бинарные деревья.

Разреженные массивы на основе бинарных деревьев

Фактически, бинарное дерево представляет собой всего лишь несколько усовершенствованную версию связного списка с двойными ссылками. Основное его преимущество по сравнению со связным списком заключается в том, что он обеспечивает быстрый поиск. Это означает, что вставка новых элементов и просмотр существующих могут осуществляться очень быстро. Если вы хотите использовать в своих приложениях структуру связного списка, но при этом требуется иметь хорошие показатели по времени поиска, то бинарные деревья — это именно то, что вам требуется. Нижеприведенная программа реализует разреженный массив, используя для хранения физического массива бинарное дерево.

```
/*
Параметризованный класс разреженного массива
на основе бинарного дерева
/*

#include <iostream.h>
#include <stdlib.h>

template <class DataT> class ArrayOb {
public:
     long index; // индекс элемента массива
     DataT info; // информация
};

// Параметризованное бинарное дерево для реализации разреженных массивов
template <class DataT>
class SparseTree : public ArrayOb<DataT> {
public:
```

```
    SparseTree *left;
    SparseTree *right;
    SparseTree *root;

    SparseTree() {root = NULL;}
    void store (SparseTree *r, SparseTree *previous,
             long ix, DataT info);
    SparseTree *dtree(SparseTree *r, long ix);
    SparseTree *find(SparseTree *r, long ix);
};

// Сохранение нового элемента в массиве
template <class DataT>
    void SparseTree<DataT>::store(SparseTree *r, SparseTree
* previous,
                              long ix, DataT info)

{
    if(!r) {
        r = new SparseTree;
        if(!r) {
            cout << "Недостаточно памяти\n";
            exit(1);
        }
        r->left = NULL;
        r->right = NULL;
        r->info = info;
        r->index = ix;
        if(!root) root = r; // первый элемент
        else {
            if(ix < previous->index) previous->left = r;
            else previous->right = r;
            }
            return;
        }

        if(ix < r->right)
            store(r->left, r, ix, info);
        else
            store(r->right, r, ix, info);
}

// Удаление элемента разреженного массива
template <class DataT>
```

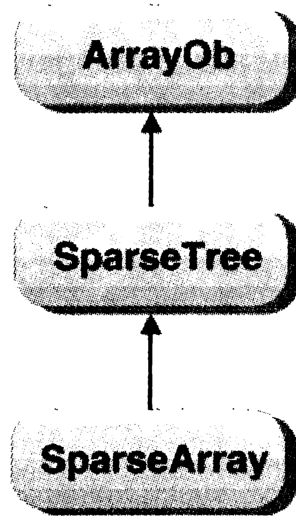
```
SparseTree<DataT> *SparseTree<DataT>::dtree(SparseTree *r,
long ix)
{
     SparseTree *p, *p2;

     if(!r) return r; // элемент не найден

     if(r->index==ix) {
          if(r->left==r->right) {
               free(r);
               if(r==root) root = NULL;
               return NULL;
               }
               else if (r->left==NULL) {
                    p = r->right;
                    free(r);
                    if(r==root) root = p;
                    return p;
               }
               else if(r->right==NULL) {
                    p = r->left;
                    free(r);
                    if(r==root) root = p;
                    return p;
               }
               else {
                    p2= r->right;
                    p = r->right;
                    while(p->left) p = p->left;
                    p->left = r->left;
                    free(r);
                    if(r==root) root = p2;
                    return p2;
               }
          }
          if(r->index <ix) r->right = dtree(r->right, ix);
          else r->left = dtree(r->left, ix);
          return r;
     }

// Поиск элемента по данному индексу

template <class DataT> SparseTree<DataT>
*SparseTree<DataT>::find(SparseTree *r, long ix);
{
```

```
     if(!r) return r;
     while(r->index != ix) {
          if(ix < r->index) r = r->left;
          else r = r->right;
          if(r==NULL) break;
          }
          return r;
     }

// Параметризованный класс разреженного массива

template <class DataT> class SparseArray : public
SparseTree<DataT> {
     long length;
public:
     SparseArray(long size) {length = size;}
     DataT &operator[ ](long I);
};

// Индексация разреженного массива

template <class DataT> DataT &SparseArray<DataT>::operator[
](long ix)
{
     if(ix<0 || ix>length-1) {
          cout << "\nИндекс ";
          cout << ix << " выходит за пределы области опреде-
ления.\n";
          exit(1);
     }

     SparseTree<DataT> *o;

     o = find(this->root, ix);
     if(!o) {
          store(this->root, NULL, ix, o);
          o = find(this->root, ix);
          }
          return o->info;
}

main()
{
```

```
    SparseArray<int> iob(100000);
    SparseArray<double> dob(100000);
    int i;

cout << "Массив целых: " << endl;
// Поместим в массив несколько чисел
for (i=0; i<5; i++) iob[i] = i;
for (i=0; i<5; i++) cout << iob[i] << " ";
cout << endl;

iob[2] = iob[3];
for (i=0; i<5; i++) cout << iob[i] << " ";
cout << endl;

iob[1000] = 9345;
iob[2000] = iob[1000] + 100;

cout << iob[1000] << " " << iob[2000] << endl;

cout << "Массив чисел с плавающей точкой: " << endl;
// Поместим в массив несколько чисел
for (i=0; i<5; i++) dob[i*100] = (double) i*1.19 ;
for (i=0; i<5; i++) cout << dob[i*100] << " ";
cout << endl;

// Попробуем получить доступ к элементу, которому значение не
присваивалось
dob[200] = dob[999];
cout << dob[200] << " " << dob[999] << endl;

return 0;
}
```

Некоторые детали реализации разреженных массивов на основе бинарных деревьев

Как и в предыдущем случае, использовавшем связные списки, так и в случае использования бинарных деревьев, при реализации разреженных массивов используется сложная многоуровневая иерархия классов-шаблонов. Как и в предыдущем случае, классом верхнего уровня является класс **ArrayOb**. Класс **ArrayOb** наследуется классом **SparseTree**, который, в свою очередь, наследуется классом **SparseArray**. Ниже приведена структурная схема этой иерархии.



Класс бинарного дерева **SparseTree** представляет собой незначительно модифицированную версию класса **tree** из главы 2. Несмотря на некоторые незначительные модификации, его логика сохранилась в неприкосновенности и не содержит по сравнению с исходной версией ничего нового. Если вам не понятны принципы работы класса **SparseTree**, вернитесь к главе 2 и внимательно прочтите ее еще раз. В целом класс **SparseArray** почти не изменился, если не считать того, что его функция **operator[ ]( )** теперь использует функции **store( )** и **find( )**, определенные в **SparseTree( )**.

Анализ метода реализации разреженных массивов на основе бинарных деревьев

При использовании бинарных деревьев поиск выполняется гораздо быстрее, чем при использовании связного списка. Не забывайте, что последовательные алгоритмы поиска в среднем требуют выполнения $n/2$ сравнений при наличии в списке n элементов. По контрасту с этим показателем, бинарный поиск в среднем требует выполнения только $\log_2 n$ сравнений при наличии в списке тех же n элементов. Однако, не следует забывать, что такие результаты могут получаться только в том случае, если дерево хорошо сбалансировано. Таким образом, следует предусмотреть механизм балансировки деревьев, которые в целом ряде приложений бывают сильно несбалансированными. Реализация этого механизма, конечно, добавит дополнительные операции и удлинит время обработки.

Разреженные массивы на основе массивов указателей

Оба метода реализации разреженных массивов — как подход с использованием связного списка, так и подход на основе бинарного дерева — требуют фазы поиска при каждой попытке доступа к элементу массива. Хотя подход с использованием бинарного дерева и является более быстрым, все же и он не обеспечивает такой скорости доступа, которая достигается при индексации обычного массива. Поэтому были разработаны еще два метода, позволяющие несколько сократить этот разрыв в скорости. Первый из исследуемых методов использует массив указателей.

Начнем наше исследование с короткого мысленного эксперимента. Представьте себе, что вам нужен массив из 1000 объектов, каждый из которых имеет длину 1000 байт. Далее, представьте себе, что этот массив не должен быть заполнен на 100%. Если вы будете декларировать этот массив обычным путем, он должен будет занять в памяти миллион байт. Однако, если декларировать массив указателей, состоящий из 1000 элементов, то такой массив займет только 4000 байт (предполагая, что адрес имеет длину 4 байта). По этой причине массив указателей потребует значительно меньшего объема памяти, чем массив, в котором хранятся сами объекты. При использовании массива указателей для хранения информации используется следующий метод. Для того, чтобы поместить в массив новый объект, сначала необходимо выделить память для этого объекта, после чего соответствующий элемент массива указателей устанавливается таким образом, чтобы указывать на этот объект. Предложенная схема обладает лучшими характеристиками производительности по сравнению с реализациями на базе связного списка и бинарного дерева. Фактически по сравнению с прямой адресацией, использующейся при реализации обычных массивов, здесь добавляется всего лишь один промежуточный уровень. Приведенная на рис. 4.1 иллюстрация показывает схему памяти при реализации разреженного массива с помощью массива указателей.

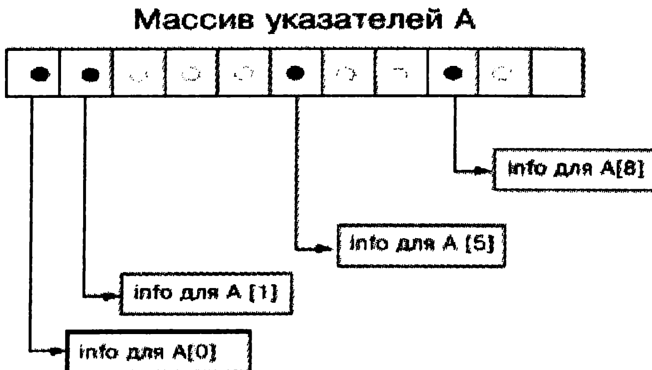


Рис. 4.1. Реализация разреженного массива на базе массива указателей

Ниже приведена версия класса **SparseArray**, использующего массив указателей.

```
/*
Параметризованный класс разреженного массива с использованием
массива указателей
*/
#include <iostream.h>
#include <stdlib.h>

/*
     Этот класс определяет тип объектов, хранящихся в разре-
женном массиве
*/
template <class DataT> class ArrayOb {
public:
     /*
     Реализация разреженного массива на базе массива указате-
     лей не требует,    чтобы индекс был членом класса ArrayOb
     */
     DataT info;
};

// Параметризованный класс разреженного массива
template <class DataT>
class SparseArray : public ArrayOb<DataT> {
     long length; // размерность массива
     ArrayOb<DataT> **ptr; // указатель на массив указателей
public:
     SparseArray(long size);
     DataT &operator[ ](long I);
};

// Конструктор для SparseArray
template <class DataT>
SparseArray<DataT>::SparseArray(long size)
{
     long i;

     length = size;

// Динамическое выделение памяти для массива указателей
     ptr = new ArrayOb<DataT> *[size];
     if(!ptr) {
          cout << "Невозможно выделить память для массива.\n";
          exit(1);
     }
```

```
// инициализация массива указателей нулем
     for(i=0; i<size; i++) {
          ptr[i] = NULL;
     }
}

// Индексация разреженного массива
template <class DataT>
DataT &SparseArray<DataT>::operator[ ](long ix)
{
     if(ix<0 || ix>length-1) {
          cout << "\nИндекс ";
          cout << ix << "за пределами границ.\n";
          exit(1);
     }
// Если элемент еще не помещен в массив, надо сделать это
if(!ptr[ix]) {
     ptr[ix] = new ArrayOb<DataT>;
     if(ptr[ix]) ptr[ix]->info = 0;
     }
     return ptr[ix]->info;
}

main()
{
     SparseArray<int> iob(1000);
     SparseArray<double> dob(1000);
     int I;

     cout << "Массив целых: " << endl;
     for(i=0; i<1000; i++) iob[i] = i;
     for(i=0; i<1000; i++) cout << iob[i] << " ";
     cout << endl;

     iob[2] = iob[3];
     iob[91] = iob[103] = 9345;
     iob[92] = iob[103]+100;
     for(i=0; i<5; i++) cout << iob[i] << " ";
     cout << endl;
     cout << iob[91] << " " << iob[92] << " " << iob[103] <<
endl;

     cout << "Массив значений с плавающей точкой: " << endl;
     for(i=0; i<5; i++) dob[i*10] = (double) i*1.19;
```

```
    for(i=0; i<5; i++) dob[i*11] = (double) I*1.29;
    for(i=0; i<5; I++)
        cout << dob[i*10] << " " << dob[i*11] << " ";
    cout << endl;

// Попробуем получить доступ к элементу, не получившему значения
    dob[20] = dob[9];
    cout << dob[20] << " " << dob[9];

    return 0;
}
```

Некоторые детали реализации разреженных массивов с помощью массива указателей

Как видно из вышеприведенной распечатки, код метода реализации разреженного массива на базе массива указателей существенно короче обоих предыдущих методов. В закрытой части класса **SparseArray** декларируется указатель **ptr** на массив указателей на объекты **ArrayOb**. Внутри конструктора **SparseArray** происходит выделение памяти для хранения массива указателей, и указатель на этот массив присваивается переменной **ptr**. Размер массива указывается параметром **size** при декларации объекта массива. Обратите внимание на то, что при построении объекта **SparseArray** все указатели в составе этого массива инициализируются значением **NULL**. Нулевой указатель обозначает, что соответствующий ему элемент массива не получил значения (то есть пуст).

Каждый раз при индексации разреженного массива функция **operator[ ]( )** просто использует один и тот же индекс для доступа к массиву указателей, возвращая ссылку на член **info** объекта **ArrayOb**, на который указывает соответствующий элемент массива указателей. Если соответствующий элемент массива указателей на данный момент имеет значение **NULL**, то создается новый объект, и указатель на этот объект сохраняется в массиве указателей.

Анализ метода, использующего массив указателей

Метод реализации разреженных массивов, построенный на базе массива указателей, обеспечивает гораздо более быстрый доступ к элементам массива, чем методы с использованием связного списка или бинарного дерева. Как уже упоминалось, на практике этот метод обеспечивает практически такую же скорость, как и адресация нормальных массивов. Далее, время доступа одинаково для всех элементов массива. При использовании методов, реализующих разреженные масси-

вы на базе связных списков или бинарных деревьев, доступ к элементам, расположенным ближе к концу, осуществляется дольше, чем к элементам, расположенным ближе к началу.

Основным недостатком метода реализации разреженных массивов на базе массива указателей является то, что он применим не для всех случаев. Во-первых, его использование оправдывает себя только в случае работы с сильно разреженными массивами. По мере того, как количество фактически используемых элементов массива будет возрастать, требования массива указателей к памяти будут приближаться к требованиям, предъявляемым обычным массивом той же размерности. Кроме того, если разреженный массив очень велик, то требования по выделению памяти могут оказаться невыполнимыми даже для массива указателей.

Хэширование

Как вы уже успели убедиться, все вышеописанные методы реализации разреженных массивов имеют как свои достоинства, так и свои недостатки. Это наводит на совершенно естественную в такой ситуации мысль: "А можно ли построить метод, сочетающий все лучшие качества уже существующих?" Из дальнейшего изложения вы увидите, что в некоторых случаях это возможно. Настоящий раздел посвящен реализации подхода к разреженным массивам, использующего комбинацию массива указателей и связного списка, и сочетающего в себе лучшие качества каждого из них.

Рассмотрим основное преимущество массива указателей — скорость. При этом основным его недостатком является то, что в памяти необходимо разместить весь этот массив. Этот недостаток становится доминирующим в случае очень больших массивов. Рассмотрим основное преимущество связного списка — экономный расход ресурсов, при котором память для хранения элементов массива выделяется только по мере надобности. Однако, если связный список содержит большое количество элементов, его производительность резко падает. Поэтому желательно было бы объединить оба этих подхода и получить алгоритм, сочетающий в себе лучшие качества обоих упомянутых методов с одновременным устранением их недостатков. Дальнейшая часть раздела обсуждает именно такой метод.

Каждый раз, когда создается разреженный массив, происходит выделение памяти под массив существенно меньшего размера (приблизительно 10% от размера оригинала). Этот небольшой массив, называемый *первичным* (primary), состоит из объектов, в которых содержатся индекс элемента, значение, ассоциированное с этим индексом, а также указатель на список переполнения. Каждый логический индекс отображается в первичный массив, и, если возможно, в нем же сохраняется информация, ассоциированная с этим логическим индексом. Поскольку (в данном примере) первичный массив составляет одну десятую от размера логического массива, на один и тот же физический индекс будут отображаться десять логических. Это указывает на возможность возникновения *конфликтов* или *колли*

зий (collisions). Если это случится, избыточный элемент будет помещен в список коллизий. Благодаря использованию этой схемы относительно слабо заполненный разреженный массив может дать неплохую производительность, поскольку большинство значений будет немедленно доступно из первичного массива. В случае возникновения коллизий ни один список не окажется длиннее 9 элементов, благодаря чему время поиска также будет очень коротким.

Для реализации этого подхода потребуется метод преобразования индекса в логическом массиве в индекс в физическом первичном массиве. Средства, с помощью которых осуществляется это преобразование, называются **хэшированием** (hashing). Хэширование представляет собой процесс вычисления индекса, по которому будет сохраняться информация по самой информации. Хэширование традиционно применялось к хранящимся на диске файлам для уменьшения времени доступа. Однако, те же самые методы можно применить и при реализации разреженных массивов. Применительно к разреженным массивам, информацией, которая должна хэшироваться, будет индекс элемента в логическом массиве. На выходе алгоритм хэширования дает индекс в физическом массиве, где фактически хранится элемент.

Алгоритм хэширования применительно к разреженным массивам может оказаться достаточно простым. Наиболее удобным методом (именно он обсуждается в данной главе) является простое деление логического индекса в пропорции соотношения размеров логического и первичного массивов. Например, если размер первичного массива составляет 10% от размера логического массива, то для получения физического индекса необходимо разделить на 10 значение логического индекса.

Для сохранения элемента в физическом массиве с использованием вышеописанного метода будет использоваться следующая процедура. Сначала логический адрес будет преобразован в физический с помощью функции хэширования. Если соответствующий элемент в физическом массиве свободен, то логический индекс и соответствующее ему значение будут сохранены там. Однако, поскольку одному физическому индексу соответствуют несколько (в нашем примере — 10) логических, возможны конфликты, или коллизии хэширования. Если такое случается, то элемент будет помещен в список коллизий. С каждым из элементов физического массива связан собственный список коллизий хэширования. Разумеется, все эти списки имеют нулевую длину до тех пор, пока не произойдет коллизия. Эта ситуация проиллюстрирована на рис. 4.2. Для того, чтобы найти элемент в физическом массиве по его логическому индексу, необходимо сначала преобразовать логический индекс в значение хэширования. После этого выполняется проверка соответствия искомого логического индекса и логического индекса элемента, физический индекс которого равен только что полученному значению функции хэширования. Если они совпадают, функция возвращает информацию. В противном случае необходимо просматривать список коллизий до тех пор, пока не будет найдено совпадение или пока не будет достигнут конец списка. Ниже приведен исходный текст программы, реализующей данный подход.

```
/*
Параметризованный класс разреженного массива, использующий
алгоритм хэширования
*/
#include <iostream.h>
#include <stdlib.h>

/*
Этот класс определяет тип объекта, хранящегося в разреженном
массиве
*/
template <class DataT> class ArrayOb {
public:
     long index; // индекс элемента массива
     DataT info; // информация
     };

// Объект разреженного массива на базе связного списка

template <class DataT>
class SparseOb: public ArrayOb<DataT> {
public:
SparseOb<DataT> *next; // указатель на следующий объект
SparseOb<DataT> *prior; // указатель на предыдущий объект

SparseOb() {
     info = 0;
     index = -1;
     next = NULL;
     prior = NULL;
     };
};

/*
Параметризованный класс связного списка с двойными ссылками
для разреженных массивов
*/
template <class DataT>
class SparseList : public SparseOb<DataT> {
     SparseOb<DataT> *start, *end;
public:
     SparseList() {start = end = NULL;}
     ~SparseList();
     void store(long ix, DataT c); // сохранение элемента
     void remove(long ix); // удаление элемента
```

```
     // возвращение указателя на элемент по заданному индексу
     SparseOb<DataT> *find(long ix);
};

// Деструктор SparseList
template <class DataT> SparseList<DataT>:: ~SparseList()
{
     SparseOb<DataT> *p, *p1;

     // освобождение всех элементов списка
     p = start;
     while(p) {
          p1 = p->next;
          delete p;
          p = p1;
          }
     }

     // Добавление элемента в список
     template <class DataT>
     void SparseList<DataT>::store(long ix, DataT c)
     {
          SparseOb<DataT> *p;

          p = new SparseOb<DataT>;
          if(!p) {
               cout << "Ошибка выделения памяти.\n";
               exit(1);
               }
               p->info = c;
               p->index = ix;

               if(start == NULL) { // первый элемент списка
                    end = start = p;
               }
               else { // добавляем в конец списка
                    p->prior = end;
                    end->next = p;
                    end = p;
                    }
               }
```

```
/*
Удаление из массива элемента с указанным индексом и обновле-
ние указателей на начало и конец
*/
template <class DataT>
void SparseList<DataT>::remove(long ix)
{
SparseOb<DataT> *ob;

ob = find(ix); // получаем указатель на элемент
if(!ob) return; // элемент не существует

if(ob->prior) {// удаляется не первый элемент
     ob->prior->next = ob->next;
     if(ob->next) { // удаляется не последний элемент
          ob->next->prior = ob->prior;
     else // удаляется последний элемент
          end = ob->prior; // обновление указателя на конец
     }
     else { // Удаляется первый элемент
          if(ob->next) { // список не пуст
               ob->next->prior = NULL;
               start = ob->next;
               }
               else // теперь список пуст
                    start = end = NULL;
          }
}

// Нахождение элемента массива по индексу

template <class DataT>
SparseOb<DataT> *SparseList<DataT>::find(long ix)
{
     SparseOb<DataT> *temp;

     temp = start;

     while(temp) {
          if(ix==temp->index) return temp; // найдено вхожде
ние
          temp = temp->next;
          }
          return NULL; // нет в списке
}
```

```
// Параметризованный класс разреженного массива
template <class DataT>
class SparseArray : public ArrayOb<DataT> {
     long length; // размерность массива
     ArrayOb<DataT> *primary;
     SparseList<DataT> *chains;
     int hash(long ix); // функция хэширования
public:
     SparseArray(long size) { length = size; }
     DataT &operator[ ](long i);
     };

// Конструктор для SparseArray с использованием хэширования
template <class DataT>
SparseArray<DataT>::SparseArray(long size)
{
     long i;

     length = size;

// Динамическое выделение памяти для первичного физического массива
primary = new ArrayOb<DataT>[hash(size)+1];
if(!primary) {
     cout << "Ошибка выделения памяти.\n";
     exit(1);
     }

// Инициализация первичного массива нулем
for(i=0; i<(hash(size)+1); i++) {
     primary[i].index = -1;
     primary[i].info = 0;
     }
     chains = new SparseList<DataT>[hash(size)+1];
}
// Индексация в разреженном массиве
template <class DataT>
DataT &SparseArray<DataT>::operator[ ](long ix)
{
     if(ix<0 || ix>length-1) {
          cout << "\nИндекс "
          cout << ix << "вышел за границы диапазона\n"
          exit(1);
          }

// Находится ли индекс в первичном массиве
if(ix == primary[hasg(ix)].index)
     return primary[hash(ix)].info;
```

```
// если индекс не находится в первичном массиве, добавить
новый элемент
if(primary[hash(ix)].index == -1){
     primary[hash(ix)].index = ix; // помещаем элемент в массив
     return primary[hash(ix)].info; // возвращаем указатель на него
     }

SparseOb<DataT> *o;

o = chains[hash(ix)].find(ix); // получаем указатель на элемент
if(!o) {
     chains[hash(ix)].store(ix, o);
     o = chains[hash(ix)].find(ix);
     }

     return o->info;
}

// Определение функции хэширования
template <class DataT>
int SparseArray<DataT>::hash(long ix)
{
     return ix/10;
}

main()
{
SparseArray<int> iob(1000);
SparseArray double dob(100);
int I;

cout << "Массив целых: " << endl;
for(i=0; i<1000; i++) iob[i] = i;
for(i=0; i<1000; i++) cout << iob[i] << " ";
cout << endl;

iob[2] = iob[3];
iob[91] = iob[103] = 9345;
iob[92] = iob[103] + 100;
for(i=0; i<5; i++) cout << iob[i] << " ";
cout << endl;
cout << iob[91] << " " << iob[92] << " " << iob[103] << endl;

cout << "Массив чисел с плавающей точкой: " << endl;
for(i=0; i<5; i++) dob[i*10] = (double) i*1.19;
for(i=0; i<5; i++) dob[i*11] = (double) I*1.29;
for(i=0; i<5; I++)
```

```
    cout << dob[i*10] << " " << dob[i*11] << " ";

    cout << endl;

// Попробуем получить доступ к несуществующему элементу
dob[20] = dob[9];
cout << dob[20] << " " << dob[9]

return 0;
}
```

Некоторые детали хэширования

Понимания принципов работы метод хэширования можно достичь путем пристального изучения класса **SparseArray**. Его листинг приведен ниже в целях удобства обсуждения.

```
// Параметризованный класс разреженного массива

template <class DataT>
class SparseArray : public ArrayOb<DataT> {
    long length; // размерность массива
    ArrayOb<DataT> *primary;
    SparseList<DataT> *chains;
    int hash(long ix); // функция хэширования
public:
    SparseArray(long size) { length = size; }
    DataT &operator[ ](long i);
    };
```

Обратите внимание на то, что были добавлены три новых закрытых члена. Первый из них - это указатель **primary**, указывающий на первичный массив меньшего размера, память для хранения которого выделяется при создании объекта **SparseArray**. Следующий закрытый член — **chains**. Он будет указывать на массив объектов **SparseList**, в котором будут содержаться списки коллизий, ассоциированные с каждым индексом первичного массива. Наконец, значение хэширования вычисляется функцией **hash( )**, приведенной ниже:

```
// Определение функции хэширования
template <class DataT>
int SparseArray<DataT>::hash(long ix)
{
    return ix/10;
}
```

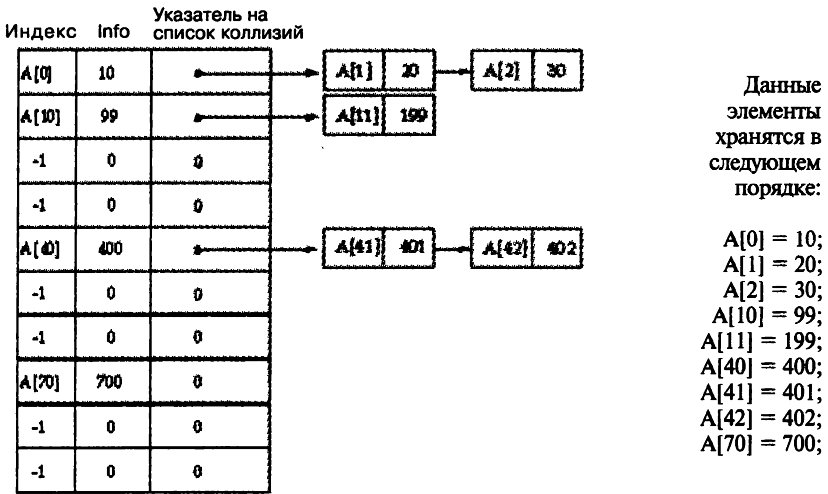


Рис. 4.2. Пример хэширования

В этом случае на один физический индекс будут отображаться 10 логических индексов. Это означает, что размер первичного массива будет составлять одну десятую часть размера логического массива. Таким образом, функция **hash( )** для получения физического индекса уменьшает логический индекс в 10 раз. Приведенная в нашем примере функция хэширования очень проста. Когда вы начнете разрабатывать собственные приложения, у вас может возникнуть потребность в построении более сложной функции хэширования.

Когда создается объект типа **SparseList**, выполняется его конструктор, листинг которого приведен ниже:

```
// Конструктор для SparseArray с использованием хэширования

template <class DataT>
SparseArray<DataT>::SparseArray(long size)
{
     long i;
     length = size;
// Динамическое выделение памяти для первичного физического массива

primary = new ArrayOb<DataT>[hash(size)+1];
if(!primary) {
     cout << "Ошибка выделения памяти.\n";
     exit(1);
     }

// Инициализация первичного массива нулем
```

```
for(i=0; i<(hash(size)+1); i++) {
     primary[i].index = -1;
     primary[i].info = 0;
     }
     chains = new SparseList<DataT>[hash(size)+1];
}
```

Эта функция выделяет первичный массив, размер которого составляет 10% от размера логического массива, и инициализирует его. Обратите внимание: вызов **hash(size)** приводит к тому, что функция **hash( )** возвращает наибольший индекс первичного массива. Добавив к этому значению 1, мы получаем требуемый размер первичного массива. После этого будет выделена память для хранения массива объектов **SparseList**.

Каждый раз, когда выполняется индексация элемента логического массива, выполняется функция **operator[ ]( )**:

```
// Индексация в разреженном массиве
template <class DataT>
DataT &SparseArray<DataT>::operator[ ](long ix)
{
     if(ix<0 || ix>length-1) {
          cout << "\nИндекс "
          cout << ix << "вышел за границы диапазона\n"
          exit(1);
          }

// Находится ли индекс в первичном массиве
if(ix == primary[hasg(ix)].index)
     return primary[hash(ix)].info;

// если индекс не находится в первичном массиве, добавить
новый элемент
if(primary[hash(ix)].index == -1){
     primary[hash(ix)].index = ix; // помещаем элемент в массив
     return primary[hash(ix)].info; // возвращаем указатель на него
     }

SparseOb<DataT> *o;

o = chains[hash(ix)].find(ix); // получаем указатель на элемент
if(!o) {
     chains[hash(ix)].store(ix, o);
```

```
    o = chains[hash(ix)].find(ix);
    }

    return o->info;
}
```

После выполнения проверки границ диапазона функция выполняет поиск индекса в первичном (физическом) массиве. Если он не присутствует в физическом массиве, и соответствующий индекс физического массива не использован (свободен), то элемент будет помещен в первичный массив. В противном случае будет выполняться поиск по списку коллизий, ассоциированному с этим индексом. Если элемент не будет найден в этом списке, то функция создаст новый элемент и поместит его в этот список. В любом случае функция возвращает ссылку на член **info**.

Анализ хэширования

В наилучшем случае (он довольно редок) каждый физический индекс, созданный хэшем, является уникальным, и время доступа приближается к времени доступа при непосредственной индексации. Это означает, что списки коллизий не создаются, и вся информация извлекается при прямом доступе. Однако, этот случай довольно-таки редок, поскольку для того, чтобы он осуществился на практике, требуется, чтобы логические индексы равномерно распределялись по пространству логических индексов. В наихудшем случае, который тоже довольно редок, схема хэширования вырождается в связный список. Это может произойти, если все хэшированные значения логических индексов будут одинаковы. Наиболее вероятен некоторый средний случай, когда метод хэширования позволит получить доступ к любому конкретному элементу в течение времени, необходимому при прямой индексации плюс некоторое значение, пропорциональное средней длине цепочек хэширования. Наиболее критичным фактором при использовании алгоритма хэширования для поддержки разреженных массивов является исключение длинных цепочек хэширования. Этого можно достигнуть только при равномерном распределении алгоритма хэширования по физическому индексу.

Выбор подхода к реализации разреженных массивов

Принимая решение об использовании для реализации разреженного массива подхода на основе связного списка, бинарного дерева, массива указателей или алгоритма хэширования необходимо учитывать такие факторы, как быстро

действие и эффективность использования памяти. Далее, вам необходимо хотя бы предположительно знать, какова будет степень заполнения реализуемого разреженного массива.

В тех случаях, когда реализуемый логический массив будет очень сильно разреженным, наибольшая эффективность использования памяти будет достигнута при использовании связных списков и бинарных деревьев, поскольку потреблять память будут только фактически используемые элементы массива. Ссылки требуют очень незначительного объема дополнительной памяти, и их влиянием, как правило, можно пренебречь. Подход на основе массива указателей требует существования всего массива в полном объеме, даже при том, что некоторые из его элементов не используются. При этом необходимо иметь такой объем памяти, который обеспечит нормальную работу вашего приложения после выделения памяти для размещения массива. Эта проблема может быть существенной для одних приложений и не иметь никакого значения для других. Как правило, вы имеете возможность вычисления объема доступной памяти и определения ее объема, необходимого для работы вашей программы. Метод хэширования занимает промежуточную позицию между подходами на основе массива указателей и подходами на основе связных списков/бинарных деревьев. Метод хэширования требует обязательного существования первичного (физического) массива даже в том случае, когда он совсем не используется. Однако, при этом объем потребной памяти для него в любом случае будет меньшим, чем для реализации на основе массива указателей.

Когда логический массив имеет высокую степень заполнения, ситуация существенным образом меняется. В этом случае более привлекательными становятся методы на основе массива указателей и хэширования. Далее, при использовании массива указателей время поиска элемента остается постоянным и не зависит от степени заполнения логического массива. Время поиска при использовании алгоритма хэширования не является постоянным, но все же оно будет ограничено неким низким значением. Что касается методов на основе связных списков и бинарных деревьев, то в них среднее время поиска увеличивается по мере заполнения массива. Этот факт следует иметь в виду, если время поиска играет существенную роль в вашем приложении.

Рекомендации для самостоятельной разработки

Классы разреженных массивов, в том виде, как они были здесь представлены, не перегружают оператор присваивания. Кроме того, они не определяют конструкторов копирования. Если вы предполагаете использовать эти классы в ситуациях, когда значения одного массива будут присваиваться другому массиву, вам придется реализовать эти операции. Если этого не сделать, то при присваивании будет иметь место побитовое копирование. Однако, по

скольку все четыре представленных здесь метода реализации разреженных массивов содержат указатели на динамически выделяемые объекты, побитовое копирование приведет к тому, что оба массива будут иметь указатели, указывающие на одни и те же элементы, и при этом внесение изменений в один из них автоматически приведет к изменению другого! Как правило, этот эффект нежелателен, и его следует избегать.

В предложенные классы можно внести и еще одну модернизацию. В некоторых приложениях требуется удалять неиспользуемые элементы. Хотя это можно сделать в индивидуальном порядке, по мере того, как элементы становятся ненужными, существует и еще один подход, который в ряде случаев может оказаться полезным и удобным. Он заключается в том, что элемент, ставший ненужным, помечается как удаленный. (Это можно реализовать, добавив к классу **ArrayOb** еще один член, определяющий статус элемента.) После этого останется только периодически выполнять функцию очистки, которая будет физически удалять все элементы, имеющие статус удаленных. Преимущество этого подхода заключается в том, что дополнительные инструкции, связанные с удалением элемента, не добавляются в код процедур программы, фактически осуществляющих доступ к элементам разреженного массива. Вместо этого операция удаления откладывается до того момента, когда процессор не загружен выполнением другой полезной работы.

Наконец, есть еще одна задача для самостоятельной разработки. Списки коллизий в алгоритме хэширования реализованы как связные списки. Рекомендуется попробовать реализовать их в виде бинарных деревьев. В большинстве случаев это обеспечит вашему приложению отличную производительность.

Глава 5

Принципы работы с информацией типа Run-time и ее использование

Одной из наиболее важных продвинутых возможностей языка C++, которую основательно знают лишь немногие, является информация RTTI (Run-Time Type Information). RTTI представляет собой подсистему, которая позволяет вашей программе распознавать типы объектов во время выполнения. RTTI не являлась частью оригинальной спецификации языка C++. Однако, ее включение в состав языка ожидалось долгое время с нетерпением, так как существует небольшой, но важный класс проблем, решение которых существенно упрощается благодаря введению RTTI. На сегодняшний день все основные компиляторы C++ поддерживают RTTI, и эта спецификация находит все более широкое применение для реализации множества программ.

Информация о типах времени выполнения (RTTI) поддерживается в C++ двумя ключевыми словами: **typeid** и **dynamic_cast**. Помимо этого, RTTI использует встроенный класс **type_info**. Большей частью RTTI поддерживается в C++ по отношению к полиморфичным классам. (Полиморфичный класс содержит как минимум одну виртуальную функцию.) Все эти функции будут подробно рассмотрены в данной главе. Кроме того, их использование будет проиллюстрировано на примерах. Наше исследование мы начнем с причин, вызывающих необходимость в присутствии RTTI.

Зачем нужна информация RTTI?

Возможно, информация о типах времени выполнения окажется для вас чем-то новым, так как ее нет в таких неполиморфичных языках программирования, как С, Pascal, BASIC или FORTRAN. В неполиморфичных языках она и не требуется, так как тип каждого объекта известен на стадии компиляции (то есть, в процессе написания программы). Однако, в полиморфичных языках, подобных С++, возможны такие ситуации, когда тип объекта на стадии компиляции неизвестен, так как точная природа объекта может быть определена только во время выполнения программы. Как вам уже должно быть известно, С++ реализует полиморфизм, используя иерархию классов, виртуальные функции и указатели на базовый класс (**base class pointers**). При таком подходе указатели на базовый класс могут использоваться для указания на объекты, принадлежащие к этому базовому классу, или на любой *объект, производный от этого базового класса.* Таким образом, заранее определить тип объекта, на который будет указывать такой указатель в каждый конкретный момент времени, не представляется возможным. Определение типа объекта должно осуществляться во время выполнения программы. В большинстве случаев это осуществляется легко и не требует RTTI. Однако, возможны и такие ситуации, когда этот метод далек от удачного. Рассмотрим ситуацию, в которой может возникнуть необходимость в RTTI.

Прежде, чем переходить к сути проблемы, сделаем следующие предположения. Предположим, что существуют полиморфичный базовый класс **B** и несколько производных от него классов. Далее предположим, что требуется написать некую функцию **F**, которая оперирует объектами типа **B** или объектами, производными от этого типа. Для того, чтобы это было возможным, **F** определяет один параметр типа **B*** (то есть, указатель базового класса). Это означает, что **F** будет воспринимать указатели типа **B&**, а также указатели на объекты типов, производных от **B**. (Не забывайте, что базовый указатель можно использовать для указания на любой объект, производный от базового.) Однако, что произойдет, если впоследствии вы (или другой ваш коллега-разработчик) определит еще один класс **D1**, также производный от **B**, в котором будут присутствовать свойства, которыми класс **B** не обладает, и при этом функция **F** должна будет принимать класс **D1** во внимание? Каким образом функция **F** будет определять, какие из объектов, на которые ей передается указатель, принадлежат к типу **D1**, а какие — нет? Иначе говоря, как функция сможет узнать, над какими из объектов, на которые указывает ее параметр, она действительно может выполнять операции? Подсистема RTTI была разработана как раз для решения таких проблем.

Для того, чтобы лучше понять существо проблемы, давайте разработаем реальный пример. В нашем примере будет использоваться простая иерархия классов для хранения координатных пар X, Y. Базовый класс этой иерархии, имеющий название **coord**, приведен ниже:

```
//Базовый полиморфичный класс
class coord {
protected:
     int x, y; // значения координат
public:
     coord() {x=0; y=o;}
     coord(int i, int j) {x=i; y=j;}
     void get_xy(int &i, int &j) {i=x; j=y;}
     void set_xy(int i, int j) {x=i; y=j;}
     virtual void show() {cout << x << "," <<y;}
};
```

На базе этого класса разработаны два производных класса, с именами **translate_coord** и **abs_coord**. Эти классы приведены ниже:

```
//Первый производный класс от coord
class translate_coord : public coord {
     int deltaX;
     int deltaY;
public:
     translate_coord() : coord() {deltaX = deltaY = 0;}
     translate_coord(int i, int j) : coord(i, j)
          {deltaX = deltaY = 0;}

     void set_trans(int a, int b)
          {deltaX = a; deltaY = b;}
     void get_trans(int &a, int &b)
          {a = deltaX, b = deltaY;}
     };

// Второй производный класс
class abs_coord : public coord {
public:
     abs_coord() : coord() {}
     abs_coord(int i, int j) : coord(abs(i), abs(j)) {}
     };
```

Класс **translate_coord** расширяет базовый класс **coord**, разрешая преобразование координат, указанное значениями **deltaX** и **deltaY**. Класс **abs_coord** создает вариацию **coord**, которая сохраняет абсолютные значения координат.

Создав эту иерархию классов, рассмотрим теперь функцию, сообщающую значение квадранта, в котором находится точка, на которую указывает **ptr**.

```
// Функция, сообщающая номер квадранта. Не совсем верна!
void quadrant(coord *ptr)
{
    int x, y;

    ptr->get_xy(x,y);

    if(x>0 && y>0)
        cout << "Точка расположена в первом квадранте\n";
    else if(x<0 && y >0)
        cout << "Точка расположена во втором квадранте\n";
    else if(x<0 && y <0)
        cout << "Точка расположена в третьем квадранте\n";
    else if(x>0 && y <0)
        cout << "Точка расположена в четвертом квадранте\n";
    else cout << "Точка расположена в начале координат\n";
}
```

На первый взгляд может показаться, что с этой функцией все в порядке. Она получает координаты указанной точки и определяет квадрант, в котором эта точка расположена. Однако, поскольку параметром этой функции является указатель **coord***, ее можно вызывать, указывая в качестве параметра указатель на объект любого из классов, производных от **coord**. Здесь-то и скрыта проблема. Например, что произойдет, если вызвать эту функцию с указателем на объект типа **translate_coord**? Как минимум, функция может неверно определить квадрант, так как она не учитывает значения преобразования **deltaX** и **deltaY**. Пытаясь устранить эту проблему, вы можете испробовать что-то вроде нижеприведенной версии функции **quadrant()**:

```
// Функция, сообщающая номер квадранта. Все еще не совсем верна!
void quadrant(coord *ptr)
{
    translate_coord *p;
    int x, y;
    int dx, dy;

    p = (translate_coord*) ptr;

    p->get_xy(x,y);
    p->get_trans(dx, dy);

    x += dx; // нормализация
    y += dy;

    if(x>0 && y>0)
```

```
            cout << "Точка расположена в первом квадранте\n";
      else if(x<0 && y >0)
            cout << "Точка расположена во втором квадранте\n";
      else if(x<0 && y <0)
            cout << "Точка расположена в третьем квадранте\n";
      else if(x>0 && y <0)
            cout << "Точка расположена в четвертом квадранте\n";
      else cout << "Точка расположена в начале координат\n";
}
```

Эта версия преобразует указатель **ptr** в указатель на объект класса **translate_coord**, получает значения преобразования координат вызовом функции **get_trans( )**, после чего нормализует координаты. Однако, теперь возникает другая проблема. Что, если функция **quadrant( )** будет вызвана с указателем на объект типа **coord** или **abs_coord**? Ни один из этих классов не определяет функцию-член **get_trans( )**. С этой программой могут происходить самые разные вещи в зависимости от вашего компилятора и вашей среды. Одним из вероятных сценариев может быть даже поломка компьютера! Совершенно ясно, что мы создали трудную ситуацию. Как же построить функцию **quadrant( )**, которая обеспечит безопасную работу со всеми типами объектов, на которые ей передается указатель? Как будет видно из дальнейшего изложения, подсистема RTTI языка C++ обеспечивает способ решения этой задачи.

Для того, чтобы иметь некую точку отсчета, приведем здесь полный листинг программы работы с координатами (которая работать не будет). В последующих разделах мы шаг за шагом рассмотрим, как можно заставить эту программу работать с помощью подсистемы RTTI.

```
// Неработающая программа, которая будет исправлена с помощью RTTI

#include <iostream.h>
#include <stdlib.h>

//Базовый полиморфичный класс
class coord {
protected:
     int x, y; // значения координат
public:
     coord() {x=0; y=o;}
     coord(int i, int j) {x=i; y=j;}
     void get_xy(int &i, int &j) {i=x; j=y;}
     void set_xy(int i, int j) {x=i; y=j;}
     virtual void show() {cout << x << "," <<y;}
};
```

```
//Первый производный класс от coord
class translate_coord : public coord {
     int deltaX;
     int deltaY;
public:
     translate_coord() : coord() {deltaX = deltaY = 0;}
     translate_coord(int i, int j) : coord(i, j)
          {deltaX = deltaY = 0;}

     void set_trans(int a, int b)
          {deltaX = a; deltaY = b;}
     void get_trans(int &a, int &b)
          {a = deltaX, b = deltaY;}
     };

// Второй производный класс
class abs_coord : public coord {
public:
     abs_coord() : coord() {}
     abs_coord(int i, int j) : coord(abs(i), abs(j)) {}
     };

/* Функция, сообщающая номер квадранта.
     *** Внимание! Она не работает ни в каком случае! ***
*/
void quadrant(coord *ptr)
{
     translate_coord *p;
     int x, y;
     int dx, dy;

     p = (translate_coord*) ptr;

     p->get_xy(x,y);
     p->get_trans(dx, dy);

     x += dx; // нормализация
     y += dy;

     if(x>0 && y>0)
          cout << "Точка расположена в первом квадранте\n";
     else if(x<0 && y >0)
          cout << "Точка расположена во втором квадранте\n";
     else if(x<0 && y <0)
          cout << "Точка расположена в третьем квадранте\n";
```

```
    else if(x>0 && y <0)
        cout << "Точка расположена в четвертом квадранте\n";
    else cout << "Точка расположена в начале координат\n";
}

main()
{
    coord coordOb(1,1), *ptr;
    translate_coord tcoordOb(1,1);
    abs_coord acoordOb(-10,-20);

    ptr = &tcoordOb;
    ptr->show();
    cout << ": ";
    quadrant(ptr); // это сработает - ptr указывает на пре-
    образуемый объект

    tcoordOb.set_trans(-10,5);
    ptr->show();
    cout << ": ";
    quadrant(ptr); // это сработает

/* ERROR - эти строки не могут выполняться
    ptr = &coordOb;
    ptr->show();
    cout << ": ";
    quadrant(ptr); // это не работает

    ptr = &acoordOb;
    ptr->show();
    cout << ": ";
    quadrant(ptr); // это не работает
*/

return 0;
}
```

Обратите внимание на то, что довольно трудно найти короткие примеры, для выполнения которых подсистема RTTI была бы необходима. Ситуации, для разрешения которых требуется RTTI, обычно связаны со сложными иерархиями классов, являющихся производными от библиотечных классов, а также со сложными взаимодействиями между приложениями, создающими и потребляющими объекты в мультизадачных средах. Во многих более простых случаях вам не понадобится прибегать к подсистеме RTTI, так как для решения проблемы будет вполне достаточно других механизмов языка C++. Это, кстати,

справедливо и в отношении вышеприведенного примера. Эту программу действительно можно привести в рабочее состояние без RTTI. (Например, **get_xy( )** можно сделать виртуальной функцией, над которой **translate_coord( )** будет иметь приоритет, и тогда преобразование координат будет выполняться автоматически.) Тем не менее, этот пример иллюстрирует существо проблемы, для решения которой и была разработана подсистема RTTI. Нижеприведенные вариации решения проблемы, использующие возможности RTTI, демонстрируют различные типы ситуаций.

Использование механизма typeid

Основной причиной, по которой наша программа обработки координат не работает, является ее неспособность к определению типа объекта, с которым оперирует функция **quadrant( )**. Как только тип объекта становится известен, проблема решается очень быстро. К счастью, в арсенале средств C++ имеется механизм **typeid**, который может определять тип объекта во время выполнения программы. Его общая форма приведена ниже:

typeid(*object*)

Здесь *object* представляет собой объект, тип которого вы будете получать. **typeid** возвращает ссылку на объект типа **object_info**, которая описывает тип объекта, определяемого параметром *object*. Для того, чтобы сделать возможным использование механизма **typeid**, в состав программы необходимо включить заголовочный файл typeinfo.h.

Класс **type_info** определяет следующие открытые члены:

```
bool_operator==(const type_info &ob) const;
bool_operator!=(const type_info &ob) const;
bool before(const type_info &ob) const;
const char *name() const;
```

Перегруженные операторы == и != обеспечивают сравнение типов. Функция **before( )** возвращает значение true, если вызывающий ее объект расположен прежде объекта, используемого в качестве параметра, в порядке следования объектов. (В основном эта функция предназначена для внутренних нужд. Возвращаемое ею значение не имеет ничего общего ни с наследованием, ни с иерархией классов.) Функция **name( )** возвращает указатель на имя типа.

Поскольку значение **typeid** оценивается во время выполнения программы, механизм **typeid** можно применять к полиморфичным типам. Это означает, что **typeid** можно использовать для определения типа объекта, на который ссылается базовый указатель полиморфичного класса. Как будет видно из дальнейшего изложения, этот факт можно использовать для исправления программы обработки координат.

Нижеприведенная программа иллюстрирует использование **typeid**:

```
// Демонстрация механизма typeid
#include <iostream.h>
#include <typeinfo.h>

class BaseClass {
     int a, b;
     virtual void f() {};
};

class Derived1: public BaseClass {
     int i, j;
};

class Derived2: public BaseClass {
     int k;
};

main()
{
     int i;
     BaseClass *p, baseob;
     derived1 ob1;
     derived2 ob2;

     // Покажем тип объекта встроенного типа
     cout << "typeid i " ;
     cout << typeid(i).name() << endl;

     // Демонстрация typeid полиморфичных типов
     p = &baseob;
     cout << "p указывает на объект типа ";
     cout << typeid(*p).name() << endl;

     p = &ob1;
     cout << "p указывает на объект типа ";
     cout << typeid(*p).name() << endl;

     p = &ob2;
     cout << "p указывает на объект типа ";
     cout << typeid(*p).name() << endl;

     // Демонстрация == и != с typeid
     if(typeid(baseob) == typeid(ob1))
          cout << "This will not be displayed.\n";
```

```
    if(typeid(baseob) != typeid(ob1))
        cout << "baseob и ob1 - объекты разных типов.\n";

    return 0;
}
```

Ниже приведен образец вывода этой программы:

```
Typeid I int
p указывает на объект типа BaseClass
p указывает на объект типа Derived1
p указывает на объект типа Derived2
baseob и ob1 - объекты разных типов.
```

Как уже упоминалось, когда **typeid** применяется к указателю базового класса полиморфичного типа, тип объекта, на который ссылается указатель, будет определяться во время выполнения программы. Важно понимать, что эта информация времени выполнения применима только к полиморфичным классам. Для эксперимента закомментируйте виртуальную функцию **f()** в классе **BaseClass** и посмотрите, что получится. **BaseClass** потеряет свой атрибут полиморфизма. В результате программа сообщит, что все объекты, на которые ссылается указатель **p**, принадлежат к типу **BaseClass**.

Использование typeid для исправления программы обработки координат

Поскольку **typeid** получает информацию о типе объекта, этот механизм можно использовать для реализации одного из подходов к исправлению нашей неработающей программы обработки координат. Текст исправленной программы приведен ниже:

```
// Использование typeid для исправления программы координат
#include <iostream.h>
#include <typeinfo.h>
#include <stdlib.h>

//Базовый полиморфичный класс
class coord {
protected:
    int x, y; // значения координат
public:
    coord() {x=0; y=0;}
    coord(int i, int j) {x=i; y=j;}
    void get_xy(int &i, int &j) {i=x; j=y;}
    void set_xy(int i, int j) {x=i; y=j;}
```

```
    virtual void show() {cout << x << "," <<y;}
};

//Первый производный класс от coord
class translate_coord : public coord {
    int deltaX;
    int deltaY;
public:
    translate_coord() : coord() {deltaX = deltaY = 0;}
    translate_coord(int i, int j) : coord(i, j)
        {deltaX = deltaY = 0;}

    void set_trans(int a, int b)
        {deltaX = a; deltaY = b;}
    void get_trans(int &a, int &b)
        {a = deltaX, b = deltaY;}
};

// Второй производный класс
class abs_coord : public coord {
public:
    abs_coord() : coord() {}
    abs_coord(int i, int j) : coord(abs(i), abs(j)) {}
    };

// Функция, сообщающая номер квадранта - версия typeid
void quadrant(coord *ptr)
{
    translate_coord *p;
    int x, y;
    int dx, dy;

    ptr->get_xy(x,y);

// Если объект преобразуется, выполнить нормализацию
if(typeid(*ptr) == typeid(translate_coord)) {
    p = (translate_coord *) ptr;
    cout << "Требуется преобразование: ";
    p->get_trans(dx,dy);
    x += dx;
    y += dy;
}

    if(x>0 && y>0)
        cout << "Точка расположена в первом квадранте\n";
```

```
    else if(x<0 && y >0)
        cout << "Точка расположена во втором квадранте\n";
    else if(x<0 && y <0)
        cout << "Точка расположена в третьем квадранте\n";
    else if(x>0 && y <0)
        cout << "Точка расположена в четвертом квадранте\n";
    else cout << "Точка расположена в начале координат\n";
}

main()
{
    coord coordOb(1,1), *ptr;
    translate_coord tcoordOb(1,1);
    abs_coord acoordOb(-10,-20);

    ptr = &tcoordOb;
    ptr->show();
    cout << ": ";
    quadrant(ptr);

    tcoordOb.set_trans(-10,5);
    ptr->show();
    cout << ": ";
    quadrant(ptr);

//Теперь эти строки могут выполняться
    ptr = &coordOb;
    ptr->show();
    cout << ": ";
    quadrant(ptr);

    ptr = &acoordOb;
    ptr->show();
    cout << ": ";
    quadrant(ptr);

    return 0;
}
```

В этой версии программы функция **quadrant()** использует **typeid** для предотвращения некорректного преобразования к типу **translate_coord**. Данная версия обеспечивает безопасность вызовов функции с параметром, который может представлять собой указатель на объект типа **coord** или любого типа, производного от

coord. Вызов функции **get_trans( )** и нормализация координат имеют место только в том случае, когда параметр **ptr** представляет собой указатель на объект типа **translate_coord**.

На данном этапе вы можете подумать, что использование RTTI в данном примере — это излишне сложное и тяжеловесное решение. Гораздо проще было бы никогда не использовать функцию **quadrant( )** с параметром, указывающим на объект типа **translate_coord** (для объектов **translate_coord** можно написать отдельную функцию). Возможно даже, что в данном конкретном случае вы и правы. Тем не менее, существуют ситуации, когда эта точка зрения неправильна. Предположим, например, что некоторая другая часть вашей программы генерирует указатели на другие типы координатных объектов, причем одни из этих объектов принадлежат к типу **coord**, другие - к типу **translate_coord**, а третьи - к типу **abs_coord**. Как в этом случае можно гарантировать, что функция **quadrant( )** никогда не будет вызвана с недопустимым типом объекта? Ключевым моментом здесь является то, что без определения типа в процессе выполнения не всегда возможно предотвратить некорректное преобразование типа указателя.

Использование механизма dynamic_cast

Хотя использование механизма **typeid**, проиллюстрированное на примере предыдущей программы, представляет собой адекватное решение проблемы, изначально возникшей с функцией **quadrant( )**, C++ предоставляет еще лучший способ реализации той же самой идеи. Этот подход использует **dynamic_cast**, один из операторов преобразования типов языка C++. Прежде, чем проиллюстрировать на примере использование оператора **dynamic_cast** для построения более совершенного решения возникшей проблемы с функцией **quadrant( )**, необходимо обзорно рассмотреть все операторы преобразования типов, существующие в языке C++.

Операторы преобразования типов

Язык C++ содержит определения пяти операторов преобразования типов. Первый из них - это стандартное преобразование, унаследованное из языка С. Вероятно, именно с этим преобразованием типов вы лучше всего знакомы (и именно оно использовалось для выполнения этого преобразования в вышеперечисленных примерах). В дополнение к этому оператору язык C++ определяет еще четыре оператора преобразования типов: **const_cast**, **dynamic_cast**, **reinterpret_cast** и **static_cast**. Базовые синтаксические конструкции с использованием этих операторов приведены ниже:

```
const_cast<type>(object)

reinterpret_cast<type>(object)
```

static_cast<*type*>(*object*)

dynamic_cast<*type*>(*object*)

Здесь *type* обозначает тип, к которому производится преобразование, а *object* — объект, преобразуемый к новому типу.

Оператор **const_cast** используется для того, чтобы явным образом отменить такие атрибуты преобразования типов, как **const** и/или **volatile**. Тип, к которому выполняется преобразование, должен быть таким же, как и исходный, за исключением отменяемых атрибутов **const** и **volatile**. Наиболее широко оператор **const_cast** используется для отмены атрибута **const**. Оператор **static_cast** выполняет неполиморфичное преобразование типов. Его можно использовать для любого стандартного преобразования. Оператор **reinterpret_cast** изменяет тип на другой, фундаментально отличающийся от исходного. Например, его можно использовать для преобразования указателя в целое. Таким образом, основное назначение оператора **reinterpret_cast** заключается в преобразовании несовместимых по наследованию типов.

Наиболее важным для RTTI является оператор **dynamic_cast**, так как он выполняет преобразование типов во время выполнения программы и проверяет допустимость выполняемого преобразования. Если преобразование выполнить невозможно, операция завершается неудачей, а результирующее выражение приравнивается к нулю. Основное назначение этого оператора заключается в выполнении преобразований над полиморфичными типами. Например, оператор **dynamic_cast** можно использовать для определения того, является ли объект, на который ссылается указатель, совместимым с конечным типом, к которому производится преобразование. При этом, если объект, на который дается указатель, не принадлежит ни к базовому и ни к одному из производных классов типа, к которому производится преобразование, выполнение оператора **dynamic_cast** завершается неудачей и выражение приравнивается к нулю. В следующем разделе мы используем этот факт для исправления функции **quadrant( )**.

Здесь будет уместно сделать следующее замечание: для удаления атрибута **const** годится только оператор **const_cast**. Таким образом, атрибут **const** не может быть удален ни одним из таких операторов, как **dynamic_cast**, **static_cast** и **reinterpret_cast**.

Использование оператора dynamic_cast для исправления функции quadrant( )

Как уже говорилось, оператор **dynamic_cast**, примененный к полиморфичному типу, будет успешно выполнен только в том случае, если преобразуемый объект или принадлежит к тому же типу, что и тип, к которому выполняется преобразование, или является производным от него. Рассмотрим, к примеру, следующее утверждение:

```
p = dynamic_cast<base*>ptr;
```

В этом случае, если **ptr** является указателем на объект типа **base** или объект, производный от **base**, то преобразование будет успешным, и **p** будет содержать указатель на объект. В противном случае преобразование завершится неудачей, и **p** будет присвоен нулевой указатель. Используя это свойство оператора **dynamic_cast**, мы можем следующим образом исправить функцию **quadrant( )**.

```
// Функция, сообщающая номер квадранта - версия dynamic_cast
void quadrant(coord *ptr)
{
     translate_coord *p;
     int x, y;
     int dx, dy;

     ptr->get_xy(x,y);

// Динамическое преобразование к типу translate_coord
p = dynamic_cast<translate_coord*>(ptr);

// Если преобразование выполнено, выполнить нормализацию
     if(p) {
          cout << "Требуется преобразование: ";
          p->get_trans(dx,dy);
          x += dx;
          y += dy;
     }

     if(x>0 && y>0)
          cout << "Точка расположена в первом квадранте\n";
     else if(x<0 && y >0)
          cout <<'"Точка расположена во втором квадранте\n";
     else if(x<0 && y <0)
          cout << "Точка расположена в третьем квадранте\n";
     else if(x>0 && y <0)
          cout << "Точка расположена в четвертом квадранте\n";
     else cout << "Точка расположена в начале координат\n";
}
```

Благодаря использованию **dynamic_cast** вы за один шаг можете выполнить операцию, которая требовала двух шагов при применении **typeid**. Версия функции **quadrant( )**, использовавшая **typeid**, требовала проверки типа объекта, на который указывает **ptr**. После этого, если этот объект принадлежал к типу **translate_coord**, выполнялось преобразование. Оператор **dynamic_cast** объединяет эти операции, выполняя их за один шаг. Именно поэтому **dynamic_cast** может решить большинство из проблем, связанных с RTTI.

Применение RTTI

Как уже упоминалось ранее, найти короткие примеры, для решения которых требовалось бы применение RTTI, довольно сложно. Тем не менее, короткая программа, разбираемая в этом разделе, поможет проиллюстрировать значение механизма RTTI. Кроме того, она даст вам понимание основных проблем, к которым применим этот механизм. Изучаемый пример создает список объектов типа **coord**. Этот список реализован как связный список указателей на объекты типа **coord**. Поскольку базовый указатель можно использовать и для ссылок на объекты производных типов, этот список можно использовать и для хранения указателей на объекты типов, производных от **coord** (в данном случае это будут типы **abs_coord** и **translate_coord**). Программа помещает указатели на объекты этих типов в список в произвольном порядке. После этого список считывается, и программа отображает квадрант, соответствующий каждому из элементов списка. Поскольку указатели хранятся в списке в произвольном порядке, узнать заранее, какие из элементов этого списка указывают на объекты типа coord, а какие — на объекты производных типов, не представляется возможным. Однако, поскольку функция **quadrant( )** использует информацию времени выполнения, для каждого из элементов будет отображаться правильное значение квадранта.

Как уже говорилось, указатели на координатные объекты будут храниться в связном списке. Для этой цели мы используем класс связного списка, разработанный в главе 2. Он будет включен в нашу программу в виде заголовочного файла с именем LIST.H. Для вашего удобства мы приведем текст класса связного списка еще раз:

```
// Параметризованный класс связного списка с двойными ссылками
#include<iostream.h>
#include<string.h>
#include<stdlib.h>

template <class DataT> class listob;

// Перегрузка << для объектов типа listob
template <class DataT>
ostream &operator<<(ostream &stream, listob<DataT> o)
{
     stream << o.info << endl;
     return stream;
}

// Перегрузка << для указателя на объект типа listob
template <class DataT>
ostream &operator<<(ostream &stream, listob<DataT> *o)
{
```

```
    stream << o->info << endl;
    return stream;
}

// Перегрузка >> для ссылки на объект типа listob
template <class DataT>
istream &operator>>(istream &stream, listob<DataT> &o)
{
    cout << "Введите информацию: ";
    stream >> o.info;
    return stream;
}

// Параметризованный класс объекта списка
template <class DataT> class listob {
public:
    DataT info; //информация
    listob<DataT> *next; //указатель на следующий объект
    listob<DataT> *prior; //указатель на предыдущий объект
    listob() {
        info = 0;
        next = NULL;
        prior = NULL;
        };
    listob(DataT c) {
        info = c;
        next = NULL;
        prior = NULL;
        }
    listob<DataT> *getnext() {return next;}
    listob<DataT> *getprior() {return prior;}
    void getinfo(DataT &c) {c = info;}
    void change(DataT c) {info =c;} //изменение элемента
    friend ostream &operator<<(ostream &stream,
    listob<DataT> o);
    friend ostream &operator<<(ostream &stream,
    listob<DataT> *o);
    friend istream &operator>>(istream &stream,
    listob<DataT> &o);
};

// Параметризованный класс списка
template <class DataT> class dllist : public listob<DataT> {
    listob<DataT> *start, *end;
public:
```

```
    dllist() {start = end = NULL;}
    ~dllist(); //деструктор для списка
    void store(DataT c);
    void remove(listob<DataT> *ob); //удаление элемента
    void frwdlist(); // отображение списка с начала
    void bkwdlist(); // отображение списка с конца

    listob<DataT> *find(DataT c); //указатель на найденное
    совпадение

    listob<DataT> *getstart() {return start;}
    listob<DataT> *getend() {return end;}
};

// Деструктор dllist
template <class DataT> dllist<DataT>::~dllist()
{
    listob<DataT> *p, *p1;

// освобождаем все элементы списка
    p = start;
    while(p) {
        p1 = p->next;
        delete p;
        p = p1;
    }
}

//Добавление нового элемента
template <class DataT> void dllist<DataT>::store(DataT c)
{
    listob<DataT> *p;

    p = new listob<DataT>;
    if(!p) {
        cout << "Ошибка выделения памяти.\n";
        exit(1);
    }

    p->info = c;

    if(start==NULL) {//первый элемент списка
        end = start = p;
    }
    else {
```

```
            p->prior = end;
            end->next = p;
            end = p;
        }
}

//Удаление элемента из списка с обновлением указателей на
начало и конец

template <class DataT> void
dllist<DataT>::remove(listob<DataT> *ob>
{
    if(ob->prior) { //не первый элемент
        ob->prior->next = ob->next;
        if(ob->next) // не последний элемент
            ob->next->prior = ob->prior;
        else // в противном случае, удаляется последний элемент
            end = ob->prior //обновление указателя на конец списка
        }
    else { //удаляется первый элемент списка
        if(ob->next) { //список не пуст
            ob->next->prior = NULL;
            start = ob->next;
        }
        else // теперь список пуст
            start = end = NULL;
    }
}

//Просмотр списка от начала к концу
template <class DataT> void dllist<DataT>::frwdlist()
{
    listob<DataT> *temp;

    temp = start;
    while(temp) {
        cout << temp->info << " ";
        temp = temp->getnext();
    }
    cout << endl;
}

// Просмотр списка в обратном направлении
template <class DataT> void dllist<DataT>::bkwdlist()
{
```

```
	listob<DataT> *temp;

	temp = end;
	while(temp) {
		cout << temp->info << " ";
		temp = temp->getprior();
	}
	cout << endl;
}

// Поиск объекта, содержащего информацию, совпадающую с указанной
template <class DataT> listob<DataT>
*dllist<DataT>::find(DataT c)
{
	listob<DataT> *temp;

	temp = start;

	while(temp) {
		if(c==temp->info) return temp; // найдено совпадение
		temp = temp->getnext();
	}
	return NULL; // совпадений не найдено
}
```

Ниже приведена программа обработки координат, создающая связный список координатных объектов:

```
// Демонстрация RTTI
#include <iostream.h>
#include <typeinfo.h>
#include <stdlib.h>
#include "list.h"

// Базовый полиморфичный класс
class coord {
protected:
	int x, y; // значения координат
public:
	coord() {x=0; y=o;}
	coord(int i, int j) {x=i; y=j;}
	void get_xy(int &i, int &j) {i=x; j=y;}
	void set_xy(int i, int j) {x=i; y=j;}
	virtual void show() {cout << x << "," <<y;}
};
//Первый производный класс от coord
```

```
class translate_coord : public coord {
     int deltaX;
     int deltaY;
public:

     };

// Второй производный класс
class abs_coord : public coord {
public:
     abs_coord() : coord() {}
     abs_coord(int i, int j) : coord(abs(i), abs(j)) {}
     };

// Функция, сообщающая номер квадранта - версия dynamic_cast
void quadrant(coord *ptr)
{
     translate_coord *p;
     int x, y;
     int dx, dy;

     ptr->get_xy(x,y);

// Динамическое преобразование к типу translate_coord
p = dynamic_cast<translate_coord*>(ptr);

// Если преобразование выполнено, выполнить нормализацию
if(p) {
     cout << "Требуется преобразование: ";
     p->get_trans(dx,dy);
     x += dx;
     y += dy;
     }

     if(x>0 && y>0)
          cout << "Точка расположена в первом квадранте\n";
     else if(x<0 && y >0)
          cout << "Точка расположена во втором квадранте\n";
     else if(x<0 && y <0)
          cout << "Точка расположена в третьем квадранте\n";
     else if(x>0 && y <0)
          cout << "Точка расположена в четвертом квадранте\n";
     else cout << "Точка расположена в начале координат\n";
}
main()
```

```
{
coord *ptr;
dllist<coord *> list; // создается список указателей на coord
listob<coord *> *p;
int i,j,x,y;

// Сохранение в списке случайным образом сгенерированных объектов
for(i=0; i<20; i++) {
     x = (rand() %500) - 250; // генерация случайных координат
     y = (rand() %500) - 250;
     j = rand() % 3; // случайным образом определяем тип объекта
     switch(j) {
          case 0:
               ptr = new coord(x,y);
               list.store(ptr);
               break;
          case 1:
               ptr = new translate_coord(x,y);
               dynamic_cast<translate_coord *>(ptr)-
>set_trans(100, 100);
               list.store(ptr);
               break;
          case 2:
               ptr = new abs_coord(x,y);
               list.store(ptr);
               break;
          }
     }

// Получение номера квадранта для каждого из элементов списка
p = list.getstart();

     while(p) {
          p->getinfo(ptr); // получаем указатель на координатный
          объект
          ptr->get_xy(x,y);
          ptr->show();
          cout << ": ";
          quadrant(ptr); // сообщаем квадрант
          p = p->getnext(); // получаем следующий объект
     }
     return 0;
}
```

Внимательно изучив функцию **main( )**, вы увидите, что различные типы указателей помещаются в список случайным образом. Это достигается благодаря использованию стандартной библиотечной функции **rand( )**. После считывания списка программа отображает координаты каждого объекта, на который дает ссылку соответствующий указатель. Таким образом, даже несмотря на то, что список содержит указатели различных типов в заранее неизвестном порядке, функция **quadrant( )** все равно может использоваться для отображения квадранта, соответствующего каждому из элементов списка.

Рекомендации для самостоятельной разработки

Как уже говорилось, механизм RTTI исключительно полезен в ситуациях, связанных со сложными иерархиями полиморфичных классов, когда заранее невозможно точно знать, на объект какого типа указывает каждый конкретный указатель в конкретный момент времени. Достаточно редкие на сегодняшний день, эти ситуации могут в будущем могут стать довольно распространенными. Например, по мере того, как растет популярность таких многопоточных многозадачных сред, как Windows 95, все чаще будут встречаться такие программы, в которых одна часть генерирует объекты, а другая использует их в некотором асинхронном режиме. Программы, в основе которых лежит архитектура этого типа, как правило, в той или иной форме будут использовать RTTI. Наконец, зарождается новая технология построения интеллектуальных объектов (smart objects), которые в своей работе будут полагаться на информацию времени выполнения.

Не считая всего вышеизложенного, для оператора **typeid** существует еще одна хорошая область применения, которую вы можете начать использовать прямо сейчас. При отладке сложных иерархий классов этот оператор может оказаться чрезвычайно полезным, так как позволит вам точно узнавать, над каким типом выполняет операции ваша программа в каждой заданной точке. Для этой цели используйте **typeid** для получения имени класса для каждого объекта.

Глава 6

Строки: использование стандартного класса строк

Язык С++ определяет несколько стандартных классов. Некоторые из этих классов имеют дело с вводом/выводом, в то время как другие образуют стандартную библиотеку шаблонов (STL, Standard Template Library). Одним из наиболее популярных стандартных классов, определенных в С++, является класс **string**. Этот класс служит для хранения строк и манипуляций с ними. Хотя стандартные строки С++ (реализованные как оканчивающиеся нулем массивы символов) в высшей степени эффективны, пользоваться ими не всегда удобно. На протяжении последних лет многие программисты на С++ разрабатывали собственные версии класса строк. Делалось это в целях упрощения обработки строк и манипуляций с ними. Разумеется, все эти реализации отличались друг от друга. Поскольку классы строк широко распространены, комитет по стандартизации ANSI С++ принял решение раз и навсегда определить класс **string**. Класс **string** как таковой на сегодняшний день включен в проект стандарта ANSI С++.

На момент написания этой книги язык С++ все еще находится в стадии определения. Это, разумеется применимо и к классу **string**. Однако, не смотря на это, вся информация, приведенная в этой главе, является стабильной и вероятность ее изменения мала. Наконец, класс **string**, в том виде, как он описан в этой главе, поддерживается компиляторами как Borland, так и Microsoft. Разумеется, вероятность изменений все же остается, хотя она и очень мала. Поэтому рекомендуется наряду с этой главой внимательно изучить руководство программиста, поставляющееся вместе с используемым вами компилятором.

Класс **string** довольно велик и содержит большое количество разнообразных возможностей - их гораздо больше, чем можно рассмотреть в одной главе. В данной главе мы рассмотрим наиболее важные, широко используемые и стабильные элементы класса **string**, после чего на примерах изучим их применение. Однако, для начала требуется небольшое разъяснение, почему включение стандартного класса **string** является таким важным дополнением к C++.

Почему стандартный класс string включен в определение C++?

Стандартные классы просто так в состав C++ не добавляются. Каждому такому добавлению предшествует период длительных размышлений и дебатов. Включение класса **string** на первый взгляд может показаться исключением из этого правила, особенно с учетом того, что C++ уже включает в себя поддержку строк, реализованных как символьные массивы, завершающиеся нулем. Однако, эта точка зрения далека от истины, и скоро мы убедимся, почему.

Как уже говорилось, C++ не содержит встроенного типа данных **string**. Вместо этого он поддерживает строки в виде завершающихся нулем символьных массивов. Такой подход к реализации строк позволяет написать в высшей степени эффективные процедуры. Фактически эта реализация строк в C++ часто преподносится как некое достижение и одна из наиболее сильных возможностей языка. Следует отметить, что она действительно позволяет осуществлять очень быстрые манипуляции со строками, а также предоставляет программисту полный контроль над этими операциями. Однако, если такой уровень скорости и эффективности не является обязательным требованием, завершающиеся нулем символьные массивы будут далеки от идеала. Напротив, в этой ситуации они теряют большую часть своей привлекательности. Причину этого понять легко: над стандартными, завершающимися нулем строками не допускаются манипуляции с помощью операторов C++. Кроме того, они не могут являться частью нормальных выражений C++. В качестве примера рассмотрим следующий фрагмент кода:

```
char str1[80], str2[80], str3[80]
// Вы не можете выполнить такую операцию
//   str1 = "This is a test";
// и эту операцию вы тоже не можете выполнить
//   str2 = str1;
// Вместо этого вы должны использовать следующее:
strcpy(str1, "This is a test");
strcpy(str2, str1);

// Вот такая операция тоже не допускается
```

```
// str3 = str2 + str1;
// Вместо этого вы должны использовать следующее:
strcpy(str3, str2);
strcat(str3, str1);
// Наконец, это вы тоже не можете выполнить:
//   if(str2<str3)
//        cout << "str2 расположена в словаре перед str3\n";
// Вместо этого вы должны использовать:
if(strcmp(str2,str3)<0)
        cout << "str2 расположена в словаре перед str3\n";
```

Как видно из этого примера, над стандартными строками, оканчивающимися нулем, можно оперировать только с помощью библиотечных функций. Поскольку технически такие строки не являются полноправными типами данных, операторы С++ к ним неприменимы. Поэтому даже самые простые операции над строками чрезвычайно громоздки и неудобны. Именно невозможность манипулирования строками с помощью операторов С++ и явилась основной причиной разработки стандартного класса строк. Ведь при определении класса в С++ вы определяете новый тип данных, который может быть полностью интегрирован в среду С++. Это, разумеется, означает возможность перегрузки операторов по отношению к новому классу. Поэтому включение класса строк позволяет обрабатывать их так же, как и любой другой тип данных: с помощью операторов.

Существует, однако, и еще одна причина разработки стандартного класса **string**: безопасность. Неопытный или небрежный программист с легкостью может написать такую программу, в процессе выполнения которой произойдет выход за границы массива, содержащего строку, завершающуюся нулем. Рассмотрим, например, стандартную функцию **strcpy()**. Эта функция не обеспечивает проверки границ массива, в который производится копирование. Если исходный массив содержит большее количество символов, чем может поместиться в том массиве, куда выполняется копирование, то велика вероятность программной ошибки или даже краха системы. Как вы увидите далее, стандартный класс **string** обеспечивает защиту от таких ошибок.

Подводя итоги, можно сказать, что существуют три основных причины, по которым стандартный класс **string** был включен в состав С++: непротиворечивость (так как класс определяет новый тип данных), удобство (так как теперь для операций со строками можно использовать операторы С++) и безопасность (средства защиты от выхода за границы массива). Однако, несмотря на все вышесказанное, нет причин совершенно забросить использование обычных строк, завершающихся нулем. Они по-прежнему представляют собой самый эффективный подход к реализации строк. Однако, в тех случаях когда скорость не является главным критерием, использование класса **string** предоставит доступ к безопасному и полностью интегрированному в среду С++ способу управления строками.

Конструкторы строк

Класс **string** поддерживает несколько конструкторов. Ниже приведены прототипы двух наиболее часто используемых конструкторов класса строк.

```
string();
string(const char *str);
```

Первая форма создает пустой объект **string**. Вторая создает объект, который содержит строку, на которую указывает **str**. Эта форма обеспечивает преобразование стандартных строк, заканчивающихся нулем, в объекты типа **string**.

На момент написания этой книги в проект стандарта языка C++ был включен дополнительный параметр для всех конструкторов **string**. Этот параметр определяет аллокатор (*allocator*), используемый объектом. (Аллокатор представляет собой зависящую от реализации класса функцию выделения и распределения памяти.) В соответствии с действующей на момент написания книги версией стандарта ANSI C++ вышеприведенные прототипы должны иметь следующую форму:

```
string(const Allocator &=Allocator());
string(const char *str, const Allocator &=Allocator());
```

Поскольку по умолчанию параметр **Allocator** соответствует стандартному аллокатору класса **string**, его в большинстве случаев можно игнорировать. Далее, на момент написания этой книги ни одна из широко доступных реализаций класса **string** не поддерживала параметр **Allocator**. Впрочем, во всех новых версиях ожидается поддержка этой возможности.

Кроме того, класс **string** поддерживает следующий конструктор копирования:

```
string(const string &strob);
```

Этот конструктор создает объект **string** и инициализирует его, используя копию объекта, указанную параметром *strob*.

Оператор	Значение
=	Присваивание
+	Конкатенация
==	Равенство
!=	Неравенство
<	Меньше
<=	Меньше или равно
>	Больше
>=	Больше или равно
[ ]	Индексирование
<<	Вывод
>>	Ввод

Операторы класса string

Для объектов класса **string** определены следующие операторы:

Эти операторы позволяют использовать объекты **string** в нормальных выражениях и исключают необходимость в вызовах таких функций, как **strcpy()** и **strcat()**. Как правило, в выражениях можно смешивать объекты **string** и стандартные строки, завершающиеся нулем. Например, объекту **string** можно присвоить в качестве значения стандартную завершающуюся нулем строку.

Как указано в последней версии стандарта ANSI C++, для стандартного класса **string** необходим заголовочный файл **string**. Однако, на момент написания книги это имя не распознавалось ни одним из широко доступных компиляторов. Например, Borland C++ использовал заголовочный файл CSTRING.H, а Microsoft Visual C++ — файл BSTRING.H. (Следует иметь в виду, что для Visual C++ файл BSTRING.H является частью стандартной библиотеки шаблонов (STL) и не инсталлируется автоматически.)

Нижеприведенная программа иллюстрирует использование класса **string**:

```
/*Короткая демонстрация стандартного класса string

    Необходимо определить BORLAND или MICROSOFT, в зависимости от того, какой компилятор вы используете. Компилятор
    Borland требует заголовочного файла CSTRING.H, MICROSOFT
    - BSTRING.H. Другие компиляторы могут требовать других
    заголовочных файлов, и в этом случае вам следует просмотреть руководство пользователя, поставляемое с вашим
    компилятором
*/

#define BORLAND

#ifdef BORLAND
#include <cstring.h>
#endif
#ifdef MICROSOFT
#include <bstring.h>
#endif
#include <iostream.h>

main()
{
    string str1("Демонстрация строк");
    string str2("Строка два");
    string str3;
```

```
    // Присваивание
    str3 = str1;
    cout << str1 << "\n" << str3 << "\n";
    // Конкатенация
    str3 = str1 + str2;
    cout << str3 << "\n";

    // Сравнение
    if(str3>str1) cout << "str3 > str1\n";
    if(str3==str1+str2) cout << "str3 == str1+str2\n";

    // Строковому объекту можно присвоить нормальную строку
    str1 = "Это нормальная строка.\n";
    cout << str1;

    //Создадим строковой объект с помощью другого строкового
объекта
    string str4(str1);
    cout << str4;

    //Ввод строки
    cout << "Введите строку: ";
    cin >> str4;
    cout << str4;

    return 0;
```

Эта программа осуществит следующий вывод:

```
Демонстрация строк
Демонстрация строк
Демонстрация строкСтрока два
str3>str1
str3 == str1+str2
Это нормальная строка.
Это нормальная строка.
Введите строку: Привет
Привет
```

Как видите, над объектами типа **string** можно манипулировать аналогично тому, как это делается по отношению ко встроенным типам данных C++. Фактически, это основное преимущество класса **string**.

Кроме того, в вышеприведенной программе следует отметить еще одну особенность: размеры строк не указаны. Размеры объектов типа **string** устанавливаются автоматически таким образом, чтобы объект мог содержать присваиваемое ему строковое значение. Таким образом, при присваиваниях и конкатенациях результирующая строка будет увеличиваться по мере надобности в соответствии с размером новой строки. Это делает невозможным выход за границы диапазона (как это может случиться со стандартными строками). Это — еще одно преимущество объектов **string** перед стандартными строками, завершающимися нулем.

Следует обратить внимание и на то, что строковой объект типа **string** можно проиндексировать с помощью оператора **[ ]**. В результате этой операции будет возвращен либо символ, соответствующий указанному индексу, либо ссылка на этот символ (в зависимости от того, как используется этот оператор). Таким образом, использование **[ ]** предоставляет доступ к конкретным символам строки. Ниже приведена короткая программа, иллюстрирующая использование оператора **[ ]**.

```
//Иллюстрация []
#define BORLAND

#ifdef BORLAND
#include <cstring.h>
#endif
#ifdef MICROSOFT
#include <bstring.h>
#endif
#include <iostream.h>

main()
{
     string str1("This is a test");

     cout << "char at [0]: << str1[0] << endl;
     cout << "char at [3]: << str1[0] << endl;
     cout << "char at [5]: << str1[0] << endl;

     return 0;
}
```

Образец вывода этой программы приведен ниже:

```
char at [0]: T
char at [3]: s
char at [5]: I
```

Наконец, последнее замечание: оператор присваивания может иметь одну из трех нижеприведенных форм:

string &operator=(const string &*strob*);
string &operator=(const char **str*);
string &operator=(const char *ch*);

Эти формы позволяют присваивать объекту **string** следующие значения: объекты **string**, стандартные строки, заканчивающиеся нулем, или символы. Таким образом, допустимы следующие типы утверждений:

strob1 = strob2; // присваивание значения строки
strob = "standard string"; // присваивание значения стандартной строки
strob = 'c'; // присваивание значения символа

Некоторые функции-члены класса string

Большинство простых операций со строками могут быть выполнены с помощью строковых операторов. В то же время, более сложные операции выполняются с помощью функций-членов класса **string**. В данном разделе мы изучим несколько наиболее распространенных функций-членов этого класса.

Присваивание и добавление частей строк

Для того, чтобы присвоить одной строке значение другой, используется функция **assign()**. Ее прототип приведен ниже:

string assign(const string &*strob*, size_t *start*=0, size_t *num*=NPOS);

Здесь **strob** представляет собой объект **string**, который будет присвоен вызывающему (инициирующему) объекту. Если присутствует параметр **start**, он указывает индекс в объекте **strob**, начиная с которого будет выполнено присваивание. Если этот параметр опустить, то присваивание будет выполняться начиная с первого символа объекта **strob**. Если присутствует параметр **num**, он указывает количество присваиваемых символов. В противном случае присваивание заканчивается с последним символом strob. **NPOS**. представляет собой значение, которое не может быть допустимым значением счетчика, как правило, оно определяется как –1. Значение NPOS определяется в заголовочном файле, поддерживающем класс **string**. В дальнейшем вы увидите, что NPOS часто используется функциями-членами класса **string**.

Примечание: на момент написания данной книги в проекте стандарта ANSI C++ имя NPOS было заменено на **npos**. Однако, ни одна из широко доступных на сегодняшний момент реализаций не учитывает этого изменения. Кроме того, значение **npos** по прежнему определяется как –1.

Если не указаны ни параметр **num**, ни параметр **string**, то функция **assign()** скопирует целую строку и ее действие будет в точности подобно действию оператора **=**. Разумеется, для присваивания целых строк гораздо удобнее пользоваться оператором **=**. Функция **assign()** понадобится вам только для присваивания частей строк. Добавить часть строки к другой строке вы можете с помощью функции-члена **append()**. Ее прототип приведен ниже:

string append(const char &strob size_t start=0, size_t num=NPOS);

Здесь **strob** представляет собой строковой объект, который будет добавлен к вызывающему (инициирующему) объекту. Если присутствует параметр **start**, он указывает индекс в объекте **strob**, с которого должна начаться конкатенация. В противном случае конкатенация начнется с первого символа объекта **strob**. Если присутствует параметр **num**, он указывает количество добавляемых символов. Если этот параметр опустить, конкатенация завершится с последним символом объекта **strob**.

Если отсутствуют как параметр **start**, так и параметр **num**, то функция **assign()** выполнит обычную конкатенацию целых строк, и ее действие будет абсолютно аналогично оператору +. Таким образом, функция **append()** нужна только для конкатенации частей строк.

Как правило, функции **assign()** и **append()** допускается вызывать, указывая в качестве первого параметра стандартную строку, завершающуюся нулем. Такие аргументы автоматически будут преобразованы в объекты класса **string**.

Нижеприведенная программа демонстрирует использование функций **assign()** и **append()**.

```
// Демонстрация функций assign() и append()
#define BORLAND

#ifdef BORLAND
#include <cstring.h>
#endif
#ifdef MICROSOFT
#include <bstring.h>
#endif
#include <iostream.h>

main()
{
    string str1("This is a test");
    string str2;

    str2.assign(str1, 5, 6)
    cout << str2 << endl;
```

```
    str2.assign("123456789", 3, 6);
    cout << str2 << endl;

    str2 = "ABCDEFG";
    str1.append(str2, 3, 4)
    cout << str1 << endl;
    str2.append("normal string", 0, 10);
    cout << str2 << endl;

    return 0;
}
```

Вывод этой программы приведен ниже:

```
is a t
456789
This is a testDEFG
ABCDEFGnormal str
```

Вставка, удаление и замена

Класс **string** определяет три типа функций-членов, которые могут изменять символы, содержащиеся в строке. Это — функции **insert()**, **remove()** и **replace()**. Прототипы их наиболее известных форм приведены ниже:

string &insert(size_t *start*, const string &*strob*,
size_t *InsertStart*=0, size_t *InsertNum*=NPOS);
string &remove(size_t *start*, size_t *num*=NPOS);
string &replace(size_t *OrgStart*, size_t *OrgNum*, const string &*strob*,
size_t *ReplaceStart*=0, size_t *ReplaceNum*=NPOS);

Функция **insert()** вставляет строку, указанную параметром **strob**, в инициировавшую вызов строку, начиная с позиции, указанной параметром **start**. Если указаны параметры **InsertStart** и **InsertNum**, то вставлены будут **InsertNum** символов, начиная с позиции **InsertStart** в объекте **strob**.

Функция **remove()** удаляет из строки-инициатора **num** символов, начиная с позиции, задаваемой параметром **start**. Если параметр **num** не указан, будут удалены все символы, начиная с позиции **start**.

Функция **replace()** заменяет **OrgNum** символов из инициирующей вызов строки, начиная с позиции **OrgStart**, строкой, задаваемой параметром **strob**. Если указаны параметры **ReplaceStart** и **ReplaceNum**, то скопированы будут только **ReplaceNum** символов объекта **strob**, начиная с позиции **ReplaceStart**.

Все эти функции демонстрирует нижеприведенная программа:

```
// Демонстрация функций insert(), remove() и replace()
#define BORLAND

#ifdef BORLAND
#include <cstring.h>
#endif
#ifdef MICROSOFT
#include <bstring.h>
#endif
#include <iostream.h>

main()
{
    string str1("This is a test");
    string str2("ABCDEFG");

    cout << "Исходные строки:\n";
    cout << "str1: " << str1 << endl;
    cout << "str2: " << str2 << "\n\n";

    // Демонстрация функции insert()
    cout << "Вставим str2 в str1:\n";
    str1.insert(5, str2);
    cout << str1 << "\n\n";

    // Демонстрация функции remove()
    cout << "Удалим 7 символов из str1:\n";
    str1.remove(5, 7);
    cout << str1 << "\n\n";

    // Демонстрация функции replace()
    cout << "Заменим 2 символа в str1 на str2:\n";
    str1.replace(5, 2, str2);
    cout << str1 << "\n\n";

    return 0;
}
```

Вывод этой программы приведен ниже:

Исходные строки:
str1: This is a test
str2: ABCDEFG

Вставим str2 в str1:
This ABCDEFGis a test

Удалим 7 символов из str1:
This is a test

Заменим 2 символа из str1 на str2:
This ABCDEFG a test

Поиск подстрок

Класс **string** предоставляет несколько функций-членов, выполняющих поиск строки, в том числе **find()** и **rfind()**. Ниже приведены прототипы для наиболее распространенных версий этих функций:

size_t find(const string &*strob*, size_t *start*=0);
size_t rfind(const string &*strob*, size_t *max*=NPOS);

Функция **find()**, начиная с позиции **start**, ищет в строке-инициаторе первое вхождение строки, содержащейся в объекте **strob**. Если эта строка будет найдена, функция возвратит индекс, соответствующий позиции строки-инициатора, на которой найдено совпадение. Если совпадений не найдено, функция возвратит значение **NPOS**.

Функция **rfind()** противоположна функции **find()**. Она просматривает строку-инициатор до позиции max и ищет последнее вхождение строки, содержащейся в **strob**. Если значение max не указано, будет выполнен поиск по всей строке. Если совпадение будет найдено, функция возвратит значение индекса, соответствующее позиции, в которой найдено совпадение, в противном случае она возвратит значение **NPOS**.

Если допускается значение параметра *start* по умолчанию, то поиск начинается с начала строки.

Нижеприведенная программа иллюстрирует использование функций **find()** и **rfind()**:

```
 Демонстрация функций find() и rfind()
#define BORLAND

#ifdef BORLAND
#include <cstring.h>
#endif
#ifdef MICROSOFT
#include <bstring.h>
```

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

main()
{
     string str1("This is a test");
     string str2("is");
     int i;

     i=str1.find(str2);
     cout << "Функция find() нашла подстроку в позиции: ";
     cout << i << endl;

     i=str1.rfind(str2);
     cout << "Функция rfind() нашла подстроку в позиции: ";
     cout << i << endl;

     return 0;
}
```

Ниже приведен вывод этой программы:

Функция find() нашла подстроку в позиции: 2
Функция rfind() нашла подстроку в позиции: 5

Как можно предположить, функцию **find()** довольно просто и удобно использовать в сочетании с функцией **replace()** для выполнения замены подстрок. Например, нижеприведенная программа заменяет все вхождения подстроки "is" на "XXX":

```
// Демонстрация функций find() и rfind()
#define BORLAND

#ifdef BORLAND
#include <cstring.h>
#endif
#ifdef MICROSOFT
#include <bstring.h>
#endif
#include <iostream.h>

main()
{
     string str1("This is a test");
     string str2("is");
     size_t I;
```

```
    while ((i=str1.find(str2)) != NPOS) {
        str1.replace(i, 2, "XXX");
        cout << "Замена в позиции " << i << endl;
        cout << "str1 теперь выглядит так: " << str1 <<
endl;
    }
    return 0;
}
```

Вывод этой программы приведен ниже:

Замена в позиции 2
str1 теперь выглядит так: ThXXX is a test
Замена в позиции 6
str1 теперь выглядит так: ThXXX XXX a test

Сравнение частей строк

Как правило, для сравнения содержимого одного объекта **string** с содержимым другого такого же объекта используются ранее описанные перегруженные реляционные операторы. Однако, если требуется сравнивать части строк, необходимо использовать функцию-член **compare**(), прототип которой приведен ниже:

int compare(const string *&strob*, size_t *start*=0, size_t *num*=NPOS) const;

В данном случае будет выполнено сравнение **num** символов в объекте **strob**, начиная с позиции start со строкой-инициатором вызова. Если эта строка меньше **strob**, функция возвратит значение, меньшее нуля. Если она больше **strob**, функция возвратит значение, большее нуля. В случае равенства **strob** и строки-инициатора функция возвратит нуль. Если параметры start и **num** принимают значения по умолчанию, то функция **compare()** выполняет сравнение целых строк и работает подобно стандартной библиотечной функции **strcmp()**.

Ниже приведен короткий пример, иллюстрирующий использование функции **compare()**:

```
// Демонстрация функции compare()
#define BORLAND

#ifdef BORLAND
#include <cstring.h>
#endif
#ifdef MICROSOFT
#include <bstring.h>
#endif
#include <iostream.h>
```

```
main()
{
     string str1("Demonstrating strings");
     string str2("Demonstrating strings, Again");

     // Сравнение целых строк
     if(str1.compare(str2) < 0) cout << "str1 < str2\n";

     // Сравнение только первых десяти символов:
     if(str1.compare(str2, 0, 10) < 0)
          cout << "Первые 10 символов str1 < str2\n";
     if(str1.compare(str2, 0, 10) == 0)
          cout << "Первые 10 символов str1 == str2\n";
     if(str1.compare(str2, 0, 10) > 0)
          cout << "Первые 10 символов str1 > str2\n";

     return 0;
}
```

Вывод этой программы приведен ниже:

```
str1<str2
Первые 10 символов str1 == str2
```

Получение длины строки

Длину строки (количество содержащихся в ней символов) можно получить через функцию-член **length()**, прототип которой приведен ниже:

```
size_t length() const;
```

Эта функция возвращает длину строки-инициатора.

Получение строки, завершающейся нулем

Хотя объекты **string** исключительно полезны, в некоторых ситуациях требуется получить версию строки, реализованной в виде завершающегося нулем массива символов. Например, объект **string** может использоваться для построения имени файла. Однако, для открытия файла необходимо задать указатель на стандартную, оканчивающуюся нулем строку. Специально для решения этой проблемы в классе **string** имеется функция-член **c_str()**. Ее прототип приведен ниже:

```
const char *c_str() const;
```

Эта функция возвращает указатель на стандартную завершающуюся нулем версию объекта **string**, инициировавшего вызов. Завершающаяся нулем строка не должна изменяться. Кроме того, после выполнения других операций над объектом **string** ее правильность не гарантируется.

Значимость функции **c_str()** иллюстрируется нижеприведенной программой. Эта программа определяет, существует ли в текущем каталоге исполняемая версия файла. По заданному из командной строки имени файла программа последовательно подбирает расширения исполняемых файлов, используя введенное имя. Таким образом, пользователь указывает имя файла (без расширения), а программа автоматически добавляет различные расширения исполняемых файлов. Программа, в том виде, как она здесь представлена, работает в среде Windows/DOS, где исполняемые файлы имеют расширения .EXE, .COM и .BAT. Обратите внимание на то, как использование класса **string** упрощает разнообразные манипуляции со строками.

```
/* Использование функции c_str()

    По заданному имени файла эта программа определяет, суще-
    ствует ли исполняемый файл с таким именем.

*/

#define BORLAND

#ifdef BORLAND
#include <cstring.h>
#endif
#ifdef MICROSOFT
#include <bstring.h>
#endif

#include <iostream.h>
#include <fstream.h>

// Расширения исполняемых файлов
char ext[3][4] = {
    "EXE"
    "COM"
    "BAT"
};

main(int argc, char *argv[])
{
    string fname;
    int i;
    int len;
```

```
    if(argc !=2) {
        cout << "Usage: execfile filename\n";
        return 1;
    }

    fname = argv[1];
    fname = fname + ".";
    len = fname.length();

    for(i=0; i<3; i++) {
        fname = fname + ext[i]; // добавляем следующее расширение

// Пытаемся открыть файл
    cout << "Trying " << fname << " ";
    ifstream f(fname.c_str()); // преобразуем в стандартную строку

    if(f) { // файл существует
        cout << " - Exists\n";
        f.close();
        }
        else cout << " - Not found\n";

        fname.remove(len); // удаляем расширение
    }
    return 0;
}
```

Ниже приведен пример вывода этой программы с использованием имени файла TEST при том условии, что в текущем каталоге существует файл TEST.EXE:

```
Trying TEST.EXE - Exists
Trying TEST.COM - Not found
Trying TEST.BAT - Not found
```

Каждый раз при выполнении цикла **for** к имени файла добавляется новое расширение. После этого построенное имя файла с расширением передается конструктору **ifstream**. Поскольку конструктор **ifstream** требует аргумента ***char**, необходимо использовать функцию **c_str()** для получения указателя на начало имени файла. Если файл существует, он будет успешно открыт. В этом случае программа выводит сообщение о том, что файл существует, и закрывает его. В противном случае файл не существует. В любом случае старое расширение удаляется, и процесс повторяется заново.

Простой строкоориентированный редактор, использующий класс string

Теперь, когда изучены наиболее распространенные элементы класса **string**, настало время применить его на практике. Наилучшим способом демонстрации возможностей класса **string** является разработка текстового редактора, так как он должен выполнять множество манипуляций со строками и их сравнений. В данном разделе мы разработаем простой, но вполне функциональный строкоориентированный редактор, использующий класс **string**.

Хотя на сегодняшний день большинство редакторов являются экранноориентированными, когда-то строкоориентированные редакторы были широко распространены. Хотя экранные редакторы в большинстве случаев гораздо проще в использовании, строкоориентированные редакторы все же имеют перед ними два преимущества. Во-первых, их гораздо легче создавать и они требуют гораздо меньше кодирования, чем экранные редакторы. Во-вторых, строкоориентированные редакторы могут работать в самых жестких условиях. К примеру, такой редактор может использоваться даже в том случае, когда единственным устройством ввода/вывода является старый телетайпный терминал! Кроме того, они полезны в тех случаях, когда в системе имеется минимальное окружение, например, на выделенных терминалах управления, где требуются минимальные возможности редактирования. Кроме того, они потребляют мало памяти и могут работать в таких условиях, когда никакой другой редактор работать не будет. Поэтому иметь на вооружении строкоориентированный редактор будет полезно для всякого программиста, вы ведь никогда не можете знать заранее, что и когда вам вдруг понадобится. Однако, основной причиной, по которой здесь будет рассматриваться строкоориентированный редактор, является то, что на его примере можно во всей полноте изучить использование класса **string**, не зарываясь в дебри другого, не относящегося к изучаемой теме кода.

Вероятнее всего, вы уже пользовались строкоориентированными редакторами ранее. Если это не так, то для понимания их базовых функций достаточно нижеприведенного краткого описания. Строкоориентированные редакторы работают с текстовыми строками (а не экранами). Управление ими осуществляется через команды. Например, в редакторе, рассматриваемом в данном разделе, для получения листинга файла используется команда L, для поиска строки — команда F и так далее. При использовании клавиш управления курсором, мыши или позиционирования экрана никаких команд не выполняется. Таким образом, строкоориентированные редакторы являются просто текстовыми манипуляторами, которые управляются командами.

Ниже приведен полный листинг строкоориентированного текстового редактора:

```
// Простой строкоориентированный текстовый редактор

#define BORLAND

#ifdef BORLAND
#include <cstring.h>
#endif
#ifdef MICROSOFT
#include <bstring.h>
#endif
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
// Класс строкоориентированного текстового редактора
class ledit {
     string text; // редактируемый файл
     size_t loc; // текущая позиция
     size_t end; // конец файла
public:
     ledit() {loc = end = 0;}
     void enter();
     void list(int numlines = 0);
     void findfirst(string what);
     void findnext(string what);
     void insert(string what);
     void del(string num);
     void exchange(string cmnd);
     void top() {loc = 0;}
     void bottom() {loc = end;}
     void where();
     void save(string fname);
     void load(string fname);
};

// Ввод текста
void ledit::enter()
{
     char nstr[255];
     string temp;

     for(;;) {
          cout << "> ";
          cin.getline(nstr, 255); // считывание введенной строки
          temp = nstr;
          if(temp == "*") break; // выход из режима ввода
          temp += "\n";
          text.insert(loc, temp); // вставить в файл
          loc += temp.length();
          end = text.length();
     }
}
```

```
// Вывод файла
void ledit::list(int numlines)
{
     size_t i, j;
     int linenum = 1;
     string temp;
     char str[255];
     i = loc;
     do {
          j = text.find("\n", i);
          if(j != NPOS) {
               j++;
               temp.assign(text, i, j-1);
               i += temp.length();
               cout << temp; // вывод строки текста
          }
          if(linenum==numlines) break;
          linenum++;
          if(!(linenum % 25)) {
               cout << "More? (Y/N)";
               cin.getline(str, 255);
               if(tolower(*str == 'n')) break;
               }
          } while(j != NPOS);
}
// Поиск
void ledit::findfirst(string what)
{
     size_t i;

     // Поиск первого вхождения
     i = text.find(what,0); // поиск от начала файла

     if(i != NPOS) { // найдено совпадение
          loc = i; // обновить текущую позицию
          list(1); // отобразить текст начиная с этой позиции
          }
          else cout << "Not found\n";
}

//Find Next
void ledit::findnext(string what)
{
     size_t i;

     // Поиск следующего вхождения
     i = text.find(what,loc+1); // поиск от текущей позиции

     if(i != NPOS) {
          loc = i;
```

```
            list(1);
            }
            else cout << "Not found\n";
}
// Вставка
void ledit::insert(string what)
{
      size_t i;

      text.insert(loc, what);
      end = text.length();

      i = loc;
      while(text[loc] != '\n' && loc) loc—;
      if(loc) loc++;
      list(1);
      loc = i;
}

// Удаление
void ledit::del(string num)
{
      size_t len;

      len = atoi(num.c_str());
      text.remove(loc, len);
      end = text.length();
      list(1);
}

// Замена одной строки на другую
void ledit::exchange(string cmnd)
{
      string oldstr, newstr;
      size_t i;

      i = cmnd.find("|", 0);
      oldstr.assign(cmnd, 0, i);
      i = text.find(oldstr, loc);

      if(i != NPOS) {
            loc = i;
            text.remove(i, oldstr.length());
            text.insert(i, newstr);
            end = text.length();
            list(1);
      }
      else cout << "Not Found\n";
}
```

```
// Показ текущей позиции
void ledit::where()
{
     list(1);
}
// Сохранить файл
void ledit::load(string fname)
{
     char ch;

     if(fname == "") {
          cout << "Enter filename: ";
          cin << fname;
     }

     ifstream in(fname.c_str());

     if(in) text = "";
     else {
          cout << "Cannot open file.\n";
          return;
     }

     while(!in.eof()) {
          in.get(ch);
          if(!in.eof()) text += ch;
     }

     in.close();

     loc = 0;
     end = text.length();
}

// Загрузить файл
void ledit::save(string fname)
{
     if(fname == "") {
          cout << "Введите имя файла: ";
          cin >> fname;
     }

     ofstream out(fname.c_str());
     if(out)
          out.write(text.c_str(), text.length());
     else {
          cout << "Невозможно открыть файл.\n";
          return;
     }
     out.close();
}
```

```
main()
{
     ledit EdOb;
     string cmnd;
     char nstr[255];
     string temp;
     char com;

     do {
          do { // получить следующую команду пользователя
               cout << ": ";
               cin.getline(nstr, 255);
          } while (!*nstr);
          cmnd = nstr;
          com = cmnd[0]; // сохранить первый символ
          cmnd.remove(0,1); // удалить первый символ

     switch(tolower(com)) {
          case 'f': // find
               EdOb.findfirst(cmnd);
               break;
          case 'n': // find next
               EdOb.findnext(cmnd);
               break;
          case 'i': // insert
               EdOb.insert(cmnd);
               EdOb.insert(cmnd);
               break;
          case 'x': // exchange
               EdOb.exchange(cmnd);
               break;
          case 'd': // delete
               EdOb.del(cmnd);
               break;
          case 'e': // enter
               EdOb.enter();
               break;
          case 'l': // list
               EdOb.top();
               EdOb.list();
               break;
          case 'b': // bottom of file
               EdOb.bottom();
               break;
```

```
        case 't': // top of file
            EdOb.top();
            break;
        case 's': // save file
            EdOb.save(cmnd);
            break;
        case 'r': // read file
            EdOb.load(cmnd);
            break;
        case 'w': // show current location in a file
            EdOb.where();
            break;
        case 'q': // quit
            break;
        default:
            cout << "?\n"; // неизвестная команда
        }
    } while (com != 'q')'

    return 0;
}
```

Таким образом, этот редактор распознает следующие команды:

Команда	Выполняемое действие
B	Переход в конец файла
Dnum	Удаляет num символов, начиная с текущей позиции
E	Вводит строки текста, начиная с текущей позиции. Начало режима ввода
Ftext	Ищет первое вхождение строки text
Ntext	Ищет следующее вхождение строки text
Itext	Вставляет text, начиная с текущей позиции
L	Выводит листинг файла
Q	Выход
Rfilename	Загружает файл с именем filename
Sfilename	Сохраняет файл под именем filename
T	Переход в начало файла
W	Показывает текущую позицию в файле (Where am I?)
Xold I new	Заменяет old text на new text. Разделителем служит вертикальная черта
.	Завершение режима ввода

Ниже приведен образец сеанса редактирования с помощью этого редактора:

```
:E
>This is a
>short test of
>simple line-oriented editor
>that uses strings.
>*
:L
This is a
short test of
simple line-oriented editor
that uses strings.
:Fis
is is a
:Nis
is a
:Xis|was
was a
:L
This was a
short test of
simple line-oriented editor
that uses strings
:T
:E
>This is on top line.
>*
:B
:E
>This is on the bottom line.
>*
:L
This is on top line.
This was a
short test of
simple line-oriented editor
that uses strings.
This is on the bottom line.
:Q
```

Некоторые детали работы редактора

Работа редактора достаточно прямолинейна и должна быть понятна из текста программы. Однако, необходимо добавить следующие комментарии. Строкоориентированный редактор инкапсулирован в класс **ledit**. Редактируемый текст трактуется как объект класса **text**. Индекс, характеризующий текущую позицию в редактируемом файле, содержится в переменной **loc**, а индекс конца файла — в переменной **end**. При создании объекта **ledit** оба индекса инициализируются нулем. Их изменения выполняют соответствующие функции-члены.

Каждая из команд редактора выполняется соответствующей функцией-членом. Например, при выводе листинга файла вызывается функция **list()**. Работу каждой из функций-членов легко отследить. Обратите особое внимание на способ, с помощью которого класс **string** делает манипуляции с текстом такими простыми и удобными.

Рекомендации для самостоятельной разработки

Нет лучшего способа ознакомления с классом **string**, чем эксперимент. Попробуйте написать короткие программы, реализующие ту или иную возможность, и проанализируйте полученные результаты. Обретя уверенность в работе с объектами типа **string**, замените ими все традиционные завершающиеся нулем строки в своих наработках. Во многих случаях вы наглядно убедитесь, что такая замена существенно упрощает код, обрабатывающий строки.

Возможно, вам покажется занимательным дальнейшее усовершенствование строкоориентированного редактора. Так, можно добавить к команде **L** спецификацию диапазона, чтобы команда отображала только определенные строки. В том виде, как она представлена в нашей программе, команда **L** отображает весь файл целиком. Добавьте команды + и –, которые будут перемещать текущую позицию на один символ влево или вправо. Эти команды будут полезны при внесении незначительных изменений. Наконец, попробуйте добавить нумерацию строк, с тем чтобы пользователь мог переходить к строке с заданным номером. Редактор настолько прост, что вы без труда внесете все модификации, которые сочтете нужными.

Глава 7

Шифрование и сжатие данных

Можно предположить, что если человек любит программирование, он любит и развлекаться с кодами и шифрами. Возможно, это происходит потому, что все коды связаны с алгоритмами так же, как и программы. А возможно, это — просто стремление к загадочным вещам, которые недоступны пониманию большинства людей. Практически наверняка программист испытывает глубокое удовлетворение, когда человек, мало знакомый с программированием, просматривает листинг программы и говорит что-то наподобие: “Да это просто невозможно понять!” В конце концов, сам процесс написания программы называется кодированием.

Существует две основные причины, по которым компьютерная криптография имеет такое большое значение. Наиболее очевидной из них является необходимость защитить конфиденциальные данные в сетях и многопользовательских системах. Хотя в большинстве случаев защита паролем является вполне адекватной мерой, конфиденциальные файлы часто шифруются для обеспечения еще более высокой степени защищенности. Второй, не столь очевидной причиной является необходимость в компьютерных кодах при передаче данных. Иногда передаваемая информация не представляет собой секрета. Однако, владелец этой информации может быть заинтересован в том, чтобы фактически эту информацию получали только те лица, которые за нее заплатили. В любом случае, методы цифрового кодирования стали важны сами по себе, вне зависимости от причины, по которой они используются.

Параллельно криптографии развивается и наука о сжатии данных. Как правило, сжатие данных используется с целью увеличения емкости различных устройств, предназначенных для хранения информации. Несмотря на то, что стоимость таких устройств за последнее время резко упала, потребность в размещении большего объема информации в меньшей области хранения по

прежнему сохраняет актуальность. Фактически на рынке программного обеспечения предлагаются несколько специализированных программ, предназначенных для сжатия данных.

В этой главе мы рассмотрим несколько алгоритмов шифрования и две схемы сжатия данных. Следует упомянуть, что основной целью данной главы (как и остальных глав этой книги) является иллюстрация возможностей языка C++. Именно по этой причине рассмотренные здесь алгоритмы шифрования преднамеренно упрощены, и автор считает своим долгом указать, что они не могут применяться там, где предъявляются жесткие требования к защите данных. Основное их назначение — это показать, как язык C++ может использоваться для решения проблем этого типа. Приведенные здесь программы могут послужить стартовой точкой для начала разработки ваших собственных алгоритмов шифрования. Если вам требуется стойкий ко взлому алгоритм шифрования, вам потребуются существенно более сложные методы, нежели те, которые рассматриваются здесь. Что касается алгоритмов сжатия данных, рассматриваемых в данной главе, то они тоже довольно просты. Однако, несмотря на эту простоту, они довольно эффективны в отношении текстовых данных.

Краткая история криптографии

История "секретного письма" теряется в глубине веков, так как никто с уверенностью не может сказать, когда и кто впервые применил его. Один из наиболее ранних известных образцов был найден на клинописной табличке, датируемой 1500 до н. э., и содержал зашифрованный рецепт изготовления керамической глазури. Известно, что древние греки использовали шифрование уже в 475 г до н. э., а высший класс древнего Рима широко использовал несложные шифры во время правления Юлия Цезаря. Как и многие другие интеллектуальные достижения, криптография была утрачена во время средних веков, и лишь изредка использовалась образованными монахами. Однако, с наступлением Ренессанса, искусство криптографии снова возродилось и начало процветать. Так, во Франции в эпоху правления Людовика XIV для правительственных сообщений применялся код, базировавшийся на 587 произвольно выбранных ключах.

В начале XIX века на развитие криптографии повлияли два фактора. Первым из них послужили взволновавшие воображение многих рассказы Эдгара По (например, "Золотой жук"), где описывались зашифрованные послания. Вторым фактором являлось изобретение телеграфа и кода Морзе. Код Морзе оказал особенно важное влияние еще и потому, что это было первое в истории двоичное (точки и тире) представление алфавита, получившее широкое распространение. Во время I Мировой войны во многих странах были разработаны "шифровальные машины", которые позволяли легко кодировать и декодировать текстовую информацию, используя сложные шифры. Эти ме

ханические устройства могли применять шифры высокой степени сложности. На этом этапе своей истории криптография, помимо науки о шифровании, стала и наукой о взламывании шифров.

До введения в обиход механических шифровальных устройств сложные шифры применялись нечасто, так как для кодирования и декодирования требовались значительные усилия и много времени. По этой же причине большинство кодов могло быть взломано за относительно короткое время. Однако, с началом применения шифровальных устройств искусство взламывания кодов существенно усложнилось. Несмотря на то, что применение современных компьютеров сделало взламывание кодов того времени довольно простой задачей, это нисколько не умаляет невероятного таланта Герберта Ярдли (Herbert Yardley), считающегося самым знаменитым взломщиком кодов всех времен. В 1915 году он смог взломать правительственный код США, и, не останавливаясь на достигнутом, устремился к вершине своих достижений: в 1922 году ему удалось взломать дипломатический код Японии, при этом он даже не знал японского языка! Сделать это он смог, пользуясь частотными таблицами японского языка.

Во время II Мировой войны методы шифрования еще более усложнились, и шифрованные сообщения часто генерировались с помощью шифровальных механизмов. Поскольку коды, производимые этими устройствами, были чрезвычайно сложны, наиболее распространенным методом взлома в те времена была кража шифровальных машин противника. При этом тяжелый и трудоемкий, но приносящий огромное интеллектуальное удовлетворение процесс взлома чужого кода просто обходился. Разумеется, обладание средством быстрой и легкой расшифровки вражеских сообщений давало существенное стратегическое преимущество. Признанным фактом является то, что наличие у союзников немецкой шифровальной машины оказало свое влияние на исход войны.

С появлением компьютеров и компьютерных сетей необходимость в защищенных и стойких к взлому кодах еще более возросла. Требуется не только защищать отдельные файлы на отдельных компьютерах, но и сам доступ к компьютеру должен быть управляемым. С тех пор были разработаны многие методы шифрования данных, и среди них — DES (Data Encryption Standard), традиционно считающийся стойким ко взлому. (Хотя, по последним данным, и DES также поддается взлому.) Кроме DES, широко доступны и другие методы, обеспечивающие высокую степень защиты. В этой главе мы рассмотрим несколько методов шифрования сообщений, не удовлетворяющих таким жестким требованиям.

Три основных типа шифров

Среди традиционных методов кодирования имеется два базовых типа — перестановка (transposition) и замена (substitution). Шифры, использующие перестановку, "перемешивают" символы сообщения по определенному прави

лу. Шифр замены замещает одни символы другими, но сохраняет порядок их следования в сообщении. Оба метода могут быть доведены до любой степени сложности. Кроме того, на их основе можно создать комплексный метод, сочетающий черты каждого из них. Появление компьютеров добавило к существующим двум методам еще один, называемый битовой манипуляцией (*bit manipulation*). Этот метод изменят компьютерное представление данных по определенному алгоритму.

Все три метода при желании могут использовать ключ (*key*). Как правило, ключ представляет собой строку символов, необходимую для того, чтобы декодировать сообщение. Однако, не стоит путать ключ с методом шифрования, поскольку его наличие является необходимым, но недостаточным условием успешной расшифровки сообщения. Кроме знания ключа, необходимо знать и алгоритм шифрования. Назначение ключа состоит в "персонализации" сообщения, с тем чтобы прочитать его (по крайней мере, с легкостью) могли только те, кому оно предназначено, даже несмотря на то, что применяемый для шифрования алгоритм широко известен.

На этом этапе вам необходимо научиться различать два базовых понятия — *открытый текст* (plain-text) и *шифрованный текст* (cipher-text). Сообщение, переданное открытым текстом, представляет собой читаемое сообщение, шифрованный текст представляет собой закодированную версию сообщения.

В этой главе мы рассмотрим различные методы кодирования текстовых файлов с использованием всех трех вышеописанных методов и изучим несколько коротких программ, которые шифруют текстовые файлы. Учтите, что все эти программы содержат функции **decode()** и **encode()**. Функция **decode()** инвертирует процесс кодирования, используемый для получения шифрованного текста.

Шифры замены

Шифр замены представляет собой метод шифрования сообщения путем замены одних символов другими на регулярной основе. Одной из простейших форм такого шифра является циклический сдвиг алфавита на определенное количество символов. Например, если алфавит сдвинуть на 3 символа, то вместо

abcdefghijklmnopqrstuvwxyz

получим:

defghijklmnopqrstuvwxyzabc

Таким образом, *a* превращается в *d*, *b* — в *e*, и т. д. Обратите внимание, что буквы "abc", находившиеся в начале алфавита, переместились в конец. Для того, чтобы закодировать сообщение, пользуясь этим методом, нужно просто заменить нормальный алфавит на его смещенную версию. Например, сообщение:

meet me at sunset

будет выглядеть следующим образом:

phhw ph dw vxqvhw

Нижеприведенная программа позволит вам шифровать текстовые сообщения по методу сдвига алфавита, указав букву, с которой должен начинаться смещенный алфавит:

```
// Простой шифр методом замены
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>
void encode(char *input, char *output, char start);
void decode(char *input, char *output, char start);
main(int argc, char *argv[])
{
     if(argc !=5) {
          cout << "Usage: input output encode/decode
offset\n";
          exit(1);
     }

     if(!isalpha(*argv[4])) {
          cout << "Start letter must be alphabetical
character.\n";
          exit(1);
     }

     if(toupper(*argv[3]) == 'E')
          encode(argv[1], argv[2], *argv[4]);
     else
          decode(argv[1], argv[2], *argv[4]);

     return 0;
}

// Encode
void encode(char *input, char *output, char start);
{
     int ch;
     ifstream in(input, ios::in | ios::binary);
     ofstream out(output, ios::out | ios::binary);

     if(!in) {
          cout << "Cannot open input file.\n";
```

```
        exit(1);
    }

    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

    start = tolower(start);
    start = start - 'a';
    do {
        ch = in.get();
        ch = tolower(ch);
        if(isalpha(ch)) {
            ch += start // сдвиг
            if(ch > 'z') ch -=26; // циклический сдвиг
        }
        if(!in.eof()) out.put((char) ch);
    } while (!in.eof());

    in.close();
    out.close();
}

// Decode
void decode(char *input, char *output, char start);
{
    int ch;
    ifstream in(input, ios::in | ios::binary);
    ofstream out(output, ios::out | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        exit(1);
    }

    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

    start = tolower(start);
    start = start - 'a';
    do {
        ch = in.get();
```

```
        ch = tolower(ch);
        if(isalpha(ch)) {
            ch -= start // сдвиг в обратном направлении
            if(ch < 'a') ch +=26; // циклический сдвиг
        }
        if(!in.eof()) out.put((char) ch);
    } while (!in.eof());

    in.close();
    out.close();
}
```

Для того, чтобы использовать эту программу для кодирования файла, укажите имя файла, который должен быть зашифрован, имя файла, который будет содержать закодированную версию, слово "encode" и букву, с которой должен начинаться алфавит, подвергнутый перестановке. Например, для кодирования файла с именем MESSAGE, используя сдвиг алфавита на 3 символа, в файл CODEMESS, используется следующая командная строка:

>code message codemess encode c

Для того, чтобы выполнить декодирование, используется командная строка.

code codemess message decode c

Вышеприведенная программа шифрования, основанная на постоянном сдвиге алфавита, сможет обмануть разве что совсем уже неопытного новичка. Ни для чего другого она не годится, поскольку взламывается исключительно просто. В конце концов, есть всего 26 возможных вариантов сдвига, и все их можно перебрать за сравнительно короткое время. Лучшим вариантом по сравнению с вышеприведенным является использование неупорядоченного алфавита, а не простого сдвига. Еще одним недостатком метода простого постоянного сдвига является то, что при кодировании этим методом сохраняются на своих местах пробелы между словами. Это еще более упрощает задачу взломщика, поэтому пробелы также следует кодировать. (Еще лучше будет кодировать и знаки препинания, но здесь это не будет сделано в целях простоты изложения.) Например, вы можете задать следующее соответствие строк, одна из которых содержит упорядоченный алфавит, а вторая, задающая преобразование, — его рандомизированную версию:

abcdefghijklmnopqrstuvwxyz<space>
qazwsxedcrfvtgbyhnujm ikolp

Дает ли эта рандомизированная версия существенное улучшение по сравнению с предыдущей версией, использовавшей простой постоянный сдвиг? Ответ будет утвердительным, так как теперь имеется 26! (очень большое число) способов упорядочения алфавита, а с учетом пробела это число возрастет до 27!

Приведенная далее программа использует улучшенный метод шифрования заменой, используя только что показанный рандомизированный алфавит. Если вы зашифруете с помощью этой программы сообщение

meet me at sunset

то оно будет выглядеть следующим образом:

tssjptspqjpumgusj

В общем, это уже труднее взломать.

```
// Улучшенный шифр методом замены
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>

const int SIZE = 28;

void encode(char *input, char *output);
void decode(char *input, char *output);
int find(char *s, char ch);

char sub[SIZE] = "qazwsxedcrfvtgbyhnujm ikolp"
char alphabet[SIZE] = "abcdefghijklmnopqrstuvwxyz ";

main(int argc, char *argv[])
{
    if(argc !=4) {
        cout << "Usage: input output encode/decode\n";
        exit(1);
    }

    if(toupper(*argv[3]) == 'E')
        encode(argv[1], argv[2]);
    else
        decode(argv[1], argv[2]);

    return 0;
}

// Encode
void encode(char *input, char *output);
{
    int ch;
    ifstream in(input, ios::in | ios::binary);
    ofstream out(output, ios::out | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        exit(1);
    }
```

```
     if(!out) {
          cout << "Cannot open output file.\n";
          exit(1);
     }

     do {
          ch = in.get();
          ch = tolower(ch);
          if(isalpha(ch) || ch == ' ')
               ch = sub[find(alphabet, ch)];
          if(!in.eof()) out.put((char) ch);
     } while (!in.eof());

     in.close();
     out.close();
}

// Decode
void decode(char *input, char *output);
{
     int ch;
     ifstream in(input, ios::in | ios::binary);
     ofstream out(output, ios::out | ios::binary);

     if(!in) {
          cout << "Cannot open input file.\n";
          exit(1);
     }

     if(!out) {
          cout << "Cannot open output file.\n";
          exit(1);
     }

     do {
          ch = in.get();
          ch = tolower(ch);
          if(isalpha(ch) || ch == ' ')
               ch = alphabet[find(sub,ch)];
          if(!in.eof()) out.put((char) ch);
     } while (!in.eof());

     in.close();
     out.close();
}
```

```
// Find index
find(char *s, char ch)
{
     register int t;

     for(t=0; t<SIZE; t++) if(ch == s[t]) return t;
     return -1;
}
```

Следует отметить, что даже этот улучшенный алгоритм шифрования заменой с легкостью может быть взломан при использовании частотных таблиц английского языка, в которых содержится частотная информация по каждой букве алфавита. Например, просмотрев закодированное сообщение и увидев, что наиболее часто в нем встречается буква s, можно предположить, что она заменила букву e, самую часто встречающуюся букву английского алфавита, а p должна заменять собой пробел. Так, методом последовательного перебора, может быть расшифрована и остальная часть сообщения. Далее, чем больше объем кодированного сообщения, тем проще расшифровать его с помощью частотных таблиц. Для того, чтобы замедлить процесс расшифровки сообщения взломщиком, применяющим частотные таблицы, можно воспользоваться ***шифром со множественными заменами*** (multiple substitution cipher). В этом случае одна и та же буква открытого текста не обязательно будет преобразовываться в одну и ту же букву зашифрованного сообщения. Этого можно добиться, включив второй рандомизированный алфавит и переключаясь между ними по заранее предопределенному методу. Приведенный ниже пример основан на переключении рандомизированных алфавитов после каждого пробела. (Однако, этот метод не шифрует пробелы.) В качестве второго рандомизированного алфавита используем, например, следующий вариант:

poi uytrewqasdfghjklmnbvcxz

Этот подход реализует нижеприведенная программа:

```
// Шифр с множественными заменами

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>

const int SIZE = 28;

void encode(char *input, char *output);
void decode(char *input, char *output);
int find(char *s, char ch);
char sub[SIZE] = "qazwsxedcrfvtgbyhnujm ikolp";
```

```
char sub2[SIZE] = "poi uytrewqasdfghjklmnbvcxz";
char alphabet[SIZE] = "abcdefghijklmnopqrstuvwxyz ";

main(int argc, char *argv[])
{
    if(argc !=4) {
        cout << "Usage: input output encode/decode\n";
        exit(1);
    }

    if(toupper(*argv[3]) == 'E')
        encode(argv[1], argv[2]);
    else
        decode(argv[1], argv[2]);

    return 0;
}

// Encode
void encode(char *input, char *output);
{
    int ch, change;
    ifstream in(input, ios::in | ios::binary);
    ofstream out(output, ios::out | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        exit(1);
    }

    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

    change = 1;
    do {
        ch = in.get();
        ch = tolower(ch);
        if(isalpha(ch))
            if(change)
                ch = sub[find(alphabet, ch)];
            else
                ch = sub2[find(laphabet, ch)];
        if(!in.eof()) out.put((char) ch);
        if(ch == ' ') change = !change;
    } while (!in.eof());
```

```
    in.close();
    out.close();
}

// Decode
void decode(char *input, char *output);
{
    int ch, change;
    ifstream in(input, ios::in | ios::binary);
    ofstream out(output, ios::out | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        exit(1);
    }

    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

    change = 1;
    do {
        ch = in.get();
        ch = tolower(ch);
        if(isalpha(ch))
            if(change)
                ch = alphabet[find(sub,ch)];
            else
                ch = alphabet[find(sub2,ch)];
        if(!in.eof()) out.put((char) ch);
        if(ch == ' ') change = !change;
    } while (!in.eof());

    in.close();
    out.close();
}

// Find index
find(char *s, char ch)
{
    register int t;
    for(t=0; t<SIZE; t++) if(ch == s[t]) return t;
    return -1;
}
```

Использование этой программы для кодирования сообщения

meet me at sunset

даст следующий кодированный текст:

tssj su qj kmdkul

Почему так получилось? А вот почему. Давайте рассмотрим совместно упорядоченный и два рандомизированных алфавита (назовем их R1 и R2 соответственно):

alphabet:	abcdefghijklmnopqrstuvwxyz<space>
R1:	qazwsxedcrfvtgbyhnujm ikolp
R2:	poi uytrewqasdfghjklmnbvcxz

Отсюда становится понятно, как работает эта программа. Сначала выбирается первый алфавит. Этот рандомизированный алфавит используется для шифровки слова "meet". В результате выполнения операции получаем слово "tssj". Так как за словом "meet" следует пробел, происходит переключение на второй рандомизированный алфавит, который и используется для шифровки слова "me", в результате чего получается слово "su". Следующий пробел вызывает переключение на первый рандомизированный алфавит. Этот процесс чередования продолжается до конца сообщения.

При использовании шифрования со множественными заменами взломать шифр с помощью частотных таблиц становится намного сложнее, так как при различных условиях одна и та же буква заменяется разными символами. Если хорошенько подумать, то с помощью нескольких рандомизированных алфавитов и более совершенного механизма переключения между ними можно добиться построения такого алгоритма, в результате применения которого все алфавитные символы будут появляться с одинаковой частотой. При взломе такого алгоритма частотные таблицы языка будут практически бесполезны.

Алгоритмы перестановок

Шифр перестановки (transposition cipher) представляет собой метод шифрования, при котором символы, составляющие сообщение, переставляются в соответствии с определенным алгоритмом, скрывая таким образом смысл передаваемого текста. Одними из первых применять этот алгоритм начали древние спартанцы еще в 475 г до н.э. Они применяли для этой цели устройство, называемое "скиталой". Принципиально скитала представляет собой полоску, наматываемую на цилиндр, причем текст пишется поперек. После этого полоска разматывалась и доставлялась получателю сообщения, который для расшифровки имел цилиндр такого же диаметра. Без цилиндра текст прочитать невозможно, так как буквы бу

дут неупорядочены. На практике, однако, этот метод оставляет желать лучшего, так как всегда можно иметь набор цилиндров различных диаметров и продолжать попытки до тех пор, пока не будет получен читаемый текст.

Компьютеризованную версию скиталы можно получить, если поместить сообщение в массив одним способом, а затем переписать его другим способом. Например:

```
union message {
        char s[100];
        char s2[20][5];
} skytale;
```

Инициализируем это объединение нулями, затем запишем в массив **skytale.s** строку:

meet me at sunset

которую будем просматривать как двумерный массив **skytale.s2**. Текст будет выглядеть следующим образом:

m	e	e	t	
m	e		a	t
	s	u	n	s
e	t	0	0	0
0	0	0	0	0

Затем, если читать текст по столбцам слева направо и сверху вниз, получим:

mm e...eest...e u...tan... ts...

где многоточия означают заполнение нулями. Для того, чтобы декодировать сообщение, необходимо заполнить столбцы массива **skytale.s2**, после чего отобразить массив **skytale.s** в нормальном порядке. **skytale.s** можно обрабатывать как строку, так как сообщение будет завершаться нулем. Нижеприведенная программа иллюстрирует использование этого метода для кодирования и декодирования сообщений.

```
// Шифр-скитала

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>

     union message {
          char s[100];
          char s2[20][5];
     } skytale;

void encode(char *input, char *output);
void decode(char *input, char *output);

main(int argc, char *argv[])
{
     int t;

// инициализация массива
     for(t=0; t<100; ++t) skytale.s[t] = '\0];

     if(argc !=4) {
          cout << "Usage: input output encode/decode\n";
          exit(1);
     }

     if(toupper(*argv[3]) == 'E')
          encode(argv[1], argv[2]);
     else
          decode(argv[1], argv[2]);

     return 0;
}

// Encode
void encode(char *input, char *output);
{
     int t, t2;
     ifstream in(input, ios::in | ios::binary);
     ofstream out(output, ios::out | ios::binary);

     if(!in) {
          cout << "Cannot open input file.\n";
          exit(1);
```

```
    }
    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

for(t=0; t<100; ++t) {
    skytale.s[t] = in.get();
    if(in.eof()) break;
    }
for(t=0; t<5; ++t)
    for(t2=0; t2<20; ++t2)
        out.put(skytale.s2[t2][t]);
in.close();
out.close();

// Decode
void decode(char *input, char *output);
{
    int t, t2;
    ifstream in(input, ios::in | ios::binary);
    ofstream out(output, ios::out | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        exit(1);
    }

    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

for(t=0; t<5 && !in.eof(); ++t)
    for(t2=0; t2<20 && !in.eof(); ++t2)
        skytale.s2[t2][t] = in.get();

for(t=0; t<100; ++t)
    out.put(skytale.s[t]);
in.close();
out.close();
}
```

Разумеется, существуют и другие методы получения сообщений, зашифрованных методом перестановки. Для компьютерной реализации особенно хорошо подходит метод перестановки букв в сообщении в соответствии с заданным алгорит-

мом. Например, нижеприведенная программа реализует метод пакетной перестановки букв, образующих сообщение. (В нашем примере, чтобы избежать чрезмерного усложнения программы, пакет представляет собой просто фиксированный блок символов.) Пользователь указывает размер пакета в командной строке. Программа затем выполняет перестановку символов в каждом пакете, пока не будет зашифрован весь файл.

```
// Шифр-перестановка

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>

void encode(char *input, char *output);
void decode(char *input, char *output);

main(int argc, char *argv[])
{
     int packet_size;

     if(argc !=5) {
          cout << "Usage: input output encode/decode
packet_size\n";
          exit(1);
     }

     packet_size = atoi(argv[4]);

     if(toupper(*argv[3]) == 'E')
          encode(argv[1], argv[2], packet_size);
     else
          decode(argv[1], argv[2], packet_size);

     return 0;
}
// Encode
void encode(char *input, char *output, int packet_size);
{
     char done, temp;
     int t;
     char s[256];
     ifstream in(input, ios::in | ios::binary);
     ofstream out(output, ios::out | ios::binary);
```

```
    if(!in) {
         cout << "Cannot open input file.\n";
         exit(1);
    }

    if(!out) {
         cout << "Cannot open output file.\n";
         exit(1);
    }

done = 0;
do {
    for(t=0; t<(packet_size*2); ++t) {
         s[t] = in.get();
         if(in.eof()) {
              s[t] = '\0'; // if eof then terminate
              done = 1;
         }
    }
    for(t=0; t<packet_size; t++) {
         temp = s[t];
         s[t] = s[t + packet_size];
         s[t + packet_size] = temp;
         t++;
         temp = s[t];
         s[t] = s[packet_size*2 - t]
         s[packet_size*2 -t] = temp;
    }
    for(t=0; t<packet_size*2; t++) out.put(s[t]);
    } while(!done);

    in.close();
    out.close();
}

// Decode
void decode(char *input, char *output, int packet_size);
{
    char done, temp;
    int t;
    char s[256];
    ifstream in(input, ios::in | ios::binary);
    ofstream out(output, ios::out | ios::binary);

    if(!in) {
         cout << "Cannot open input file.\n";
         exit(1);
    }
```

```
    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

done = 0;
do {
    for(t=0; t<(packet_size*2); ++t) {
        s[t] = in.get();
        if(in.eof()) {
            s[t] = '\0'; // if eof then terminate
            done = 1;
        }
    }
    for(t=0; t<packet_size; t++) {
        t++;
        temp = s[t];
        s[t] = s[packet_size*2 - t];
        s[packet_size*2 - t] = temp;
        t—;
        temp = s[t];
        s[t] = s[t + packet_size];
        s[t + packet_size] = temp;
        t++;
    }
    for(t=0; t<packet_size*2; t++) out.put(s[t]);
    } while(!done);

    in.close();
    out.close();
}
```

Если шифры перестановок используются сами по себе, а не в комбинации с другим методом, то все они имеют тот общий недостаток, что процесс перестановок зачастую самопроизвольно создает "ключи" для взлома.

Шифры битовых манипуляций

Вышеприведенные методы шифрования представляют собой компьютеризированные версии шифрования, ранее выполнявшегося вручную. Однако, компьютерные технологии дали начало новому методу кодирования сообщений путем манипуляций с битами, составляющими фактические символы нешифрованного сообщения. Как правило, современные компьютеризированные шифры попадают в класс, называемый шифрами битовых манипуляций (*bit manipulation*

ciphers). Хотя ревнители чистоты теории могут спорить о том, что такие шифры представляют собой просто вариацию шифров методом замены, большинство специалистов согласится с тем, что концепции и методы, лежащие в основе шифров битовых манипуляций отличаются от всего, что было известно ранее, настолько значительно, что заслуживают выделения в особый класс.

Шифры битовых манипуляций популярны по двум причинам. Во-первых, они идеально подходят для использования в компьютерной криптографии, так как используют операции, которые легко выполняются системой. Вторая причина заключается в том, что полученный на выходе зашифрованный текст выглядит абсолютно нечитаемым — фактически полной бессмыслицей. Это положительно сказывается на безопасности и защищенности, так как важные данные маскируются под поврежденные файлы, доступ к которым просто никому не нужен.

Как правило, шифры битовых манипуляций применимы только к компьютерным файлам и не могут использоваться для бумажных копий зашифрованных сообщений. Причина этого заключается в том, что манипуляции с битами имеют тенденцию генерировать непечатаемые символы. Поэтому мы всегда будем полагать, что текст, зашифрованный с помощью битовых манипуляций, всегда будет оставаться в виде электронного документа.

Шифры битовых манипуляций преобразуют открытый текст в зашифрованный, преобразуют набор бит каждого символа по определенному алгоритму, используя одну из следующих логических операций или их комбинацию:

AND OR NOT XOR

C++ — один из наиболее удобных языков программирования для создания шифров, манипулирующих с битами, поскольку он поддерживает поразрядные операции, используя следующие двоичные операторы:

Оператор	Значение
&	AND (Логическое "И")
|	OR (Логическое "ИЛИ")
~	NOT (1-е дополнение)
^	XOR (Исключающее "ИЛИ")

Простейший (и наименее защищенный) шифр, манипулирующий с битами, использует только оператор первого дополнения (~). Этот оператор инвертирует все биты, входящие в состав байта. Таким образом, все нули становятся единицами, а единицы — нулями. Поэтому байт, над которым дважды проведена такая операция, принимает исходное значение. Ниже приведена программа, иллюстрирующая этот подход к шифрованию файлов.

```
// Шифр на базе первого дополнения

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>

void encode(char *input, char *output);
void decode(char *input, char *output);

main(int argc, char *argv[])
{
     if(argc !=4) {
          cout << "Usage: input output encode/decode\n";
          exit(1);
     }

     if(toupper(*argv[3]) == 'E')
          encode(argv[1], argv[2]);
     else
          decode(argv[1], argv[2]);

     return 0;
}
// Encode
void encode(char *input, char *output);
{
     int ch;
     ifstream in(input, ios::in | ios::binary);
     ofstream out(output, ios::out | ios::binary);

     if(!in) {
          cout << "Cannot open input file.\n";
          exit(1);
     }

     if(!out) {
          cout << "Cannot open output file.\n";
          exit(1);
     }

     do {
          ch = in.get();
          ch = ~ch;
          if(!in.eof()) out.put((char) ch);
     } while(!in.eof());
```

```
        in.close();
        out.close();
}

// Decode
void decode(char *input, char *output);
{
        int ch;
        ifstream in(input, ios::in | ios::binary);
        ofstream out(output, ios::out | ios::binary);

        if(!in) {
                cout << "Cannot open input file.\n";
                exit(1);
        }

        if(!out) {
                cout << "Cannot open output file.\n";
                exit(1);
        }

        do {
                ch = in.get();
                ch = ~ch;
                if(!in.eof()) out.put((char) ch);
        } while(!in.eof());

        in.close();
        out.close();
}
```

К сожалению, здесь нет возможности привести пример зашифрованного текста, так как используемые битовые манипуляции, как правило, генерируют непечатаемые символы. Откомпилируйте эту программу и испытайте ее на своем компьютере — полученный результат будет выглядеть совершенно нечитаемым!

В действительности с этой простой схемой кодирования связаны две основных проблемы. Во-первых, программа шифрования для расшифровки текста не использует ключа. Поэтому любой, кто знает алгоритм и может написать такую программу, сможет прочитать файл. Во-вторых (и это самое главное), этот метод отнюдь не тайна для опытных программистов.

Улучшенный метод шифрования методом побитовой манипуляции использует оператор XOR. Результаты выполнения этого оператора приведены в следующей таблице:

^	1	0
1	0	1
0	1	0

Иными словами, результат выполнения оператора XOR получает значение TRUE тогда и только тогда, когда один из операндов имеет значение TRUE, а другой — FALSE. Именно это и является уникальным свойством оператора XOR — если вы выполните эту операцию над одним байтом, используя другой байт в качестве "ключа", а затем возьмете результат и выполните над ним ту же самую операцию с помощью того же самого ключа, вы снова получите исходный байт. Например:

Исходный байт		11011001
Ключ	^	0101 0011 (ключ)
Результат операции		10001010

Зашифрованный байт		10001010
Ключ	^	0101 0011 (ключ)
Расшифрованный байт		1101 1001

Расшифрованный байт равен исходному.

Этот процесс может использоваться для кодирования файлов, так как он решает две основных проблемы с простейшей версией на базе первого дополнения. Во-первых, благодаря использованию ключа, расшифровать файл, имея только программу декодирования, нельзя. Во-вторых, используемые манипуляции с битами не настолько просты, чтобы их можно было сразу же распознать.

Ключ не обязательно должен иметь длину 1 байт. Фактически, вы можете использовать ключ, состоящий из нескольких символов, и чередовать эти символы на протяжении всего файла. Нижеприведенная программа, однако, в целях простоты изложения, использует односимвольный ключ.

```
// Шифр на базе XOR

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>

void encode(char *input, char *output, char key);
void decode(char *input, char *output, char key);
main(int argc, char *argv[])
```

```
{
     if(argc !=5) {
          cout << "Usage: input output encode/decode key\n";
          exit(1);
     }

     if(toupper(*argv[3]) == 'E')
          encode(argv[1], argv[2], *argv[4]);
     else
          decode(argv[1], argv[2], *argv[4]);

     return 0;
}

// Encode
void encode(char *input, char *output, char key);
{
     int ch;
     ifstream in(input, ios::in | ios::binary);
     ofstream out(output, ios::out | ios::binary);

     if(!in) {
          cout << "Cannot open input file.\n";
          exit(1);
     }

     if(!out) {
          cout << "Cannot open output file.\n";
          exit(1);
     }

     do {
          ch = in.get();
          ch = ch^key;
          if(!in.eof()) out.put((char) ch);
     } while(!in.eof());

     in.close();
     out.close();
}

// Decode
void decode(char *input, char *output, char key);
{
     int ch;
     ifstream in(input, ios::in | ios::binary);
     ofstream out(output, ios::out | ios::binary);
     if(!in) {
```

```
        cout << "Cannot open input file.\n";
        exit(1);
    }

    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

    do {
        ch = in.get();
        ch = ch^key;
        if(!in.eof()) out.put((char) ch);
    } while(!in.eof());

    in.close();
    out.close();
}
```

Откомпилируйте эту программу и испытайте ее на своем компьютере. Вы увидите, что зашифрованный текст абсолютно нечитаем и может быть расшифрован только с помощью ключа.

Сжатие данных

Назначение сжатия данных заключается в том, чтобы разместить один и тот же объем информации в меньшей области хранения. Область применения сжатия данных не ограничивается только компьютерами — так, хранение информации на микрофильмах преследует ту же цель и тоже является сжатием данных. Однако, в компьютерных системах сжатие данных используется наиболее широко, так как позволяет увеличить объем области хранения данных, сократить время передачи данных (особенно по телефонным линиям) и обеспечить должный уровень защиты данных. Существует множество универсальных методов сжатия данных, таких, как кодировка Хаффмана (Huffman) и метод Лемпел-Зива (Lempel-Ziv). Оба этих метода достаточно сложны, и их рассмотрение не входит в цели, поставленные в данной главе. Вместо этого мы рассмотрим здесь два простых, но довольно эффективных метода сжатия текстовых данных. Первый метод сжимает данные из 8 бит в 7, а второй — из 4 в 3. Как вы увидите из дальнейшего изложения, эти методы, несмотря на свою простоту, обеспечивают вполне удовлетворительное сжатие чисто текстовых файлов.

Преобразование 8-битного набора символов в 7-битный

Для большинства языков полный набор символов можно представить с помощью только 6 бит. Например, английский алфавит содержит 26 букв. Таким образом, для представления заглавных и строчных букв требуется только 52 различных кода. Поскольку 6 бит позволяют закодировать значения от 0 до 63, этого более чем достаточно для представления всех символов английского языка. Фактически, имея дело с английским языком, можно закодировать даже некоторые знаки препинания. Несмотря на это, архитектура большинства компьютеров использует 8-битный байт. Таким образом, если вы работаете только с текстами, то 2 бита — целых 25% каждого байта — расходуются впустую. Таким образом, чисто теоретически можно ужать 4 байта в 3, если использовать "лишние" 2 бита от каждого байта. Единственная проблема заключается в том, что символов ASCII больше 64, и при этом алфавитные символы не используют первые 64 значения. Это означает, что коды ASCII для целого ряда символов (в особенности для строчных букв алфавита) требуют 7-битного представления. Разумеется, можно использовать другое представление, отличное от ASCII (именно эта тема рассматривается в следующем разделе), но это не всегда желательно. Однако, даже при использовании стандартной кодовой таблицы ASCII 1 бит все равно тратится впустую. Поэтому можно разработать альтернативный метод сжатия 8 бит в 7, используя тот факт, что ни одна буква или знак пунктуации не используют восьмой бит. Этот метод позволит сжать данные на 12,5%. При этом вы должны отдавать себе отчет в том, что многие компьютеры (в том числе и PC) используют восьмой бит для представления специальных символов. Кроме того, некоторые текстовые процессоры используют восьмой бит для указания инструкций по обработке текста. Поэтому описываемый здесь метод будет действовать только для стандартных ASCII-файлов, в которых восьмой бит не используется ни для какой цели.

Существует несколько способов сжатия 8 бит до 7. Для понимания подхода, использованного в примере, рассмотрим следующие восемь символов, используя стандартный 8-битный код:

байт 1: 0111 0101
байт 2: 0111 1101
байт 3: 0010 0011
байт 4: 0101 0110
байт 5: 0001 0000
байт 6: 0110 1101
байт 7: 0010 1010
байт 8: 0111 1001

Как видите, восьмой бит всегда равен нулю. Если он не используется для проверки четности, это всегда будет так. Простейшим способом уплотнить восемь символов до 7 является распределение семи значащих битов первого байта по семи незанятым позициям неиспользуемого восьмого бита всех остальных байтов (от второго до восьмого). Если проделать эту операцию, то 7 оставшихся байт будут выглядеть следующим образом

```
        ┌── Байт 1 — читается сверху вниз
        ↓
байт 2: 1111 1101
байт 3: 1010 0011
байт 4: 1101 0110
байт 5: 0001 0000
байт 6: 1110 1101
байт 7: 0010 1010
байт 8: 1111 1001
```

Для того, чтобы восстановить первый байт, необходимо составить его, используя значения восьмого бита оставшихся семи байт.

Описанный здесь метод сжатия уплотнит текстовый файл на 1/8, т. е., на 12,5%. Это уже существенная экономия. Предположим, если вы передаете программный код своему товарищу через коммутируемые телефонные линии на большое расстояние, этот алгоритм позволит вам сэкономить 12,5% оплаты за телефон.

Приведенная ниже программа сжимает текстовые файлы описанным выше методом. Следует знать, что для того, чтобы алгоритм корректно работал в конце файла, к выходному файлу могут быть добавлены до 7 дополнительных байт. Таким образом, если вы пытаетесь сжать очень короткие файлы (короче 56 байт), то сжатый файл может оказаться даже длиннее исходного. Однако, для больших файлов эти дополнительные 7 байт значения не имеют. Есть интересная задача — попробуйте усовершенствовать этот алгоритм так, чтобы дополнительные 7 байт стали ненужными.

```
// Сжатие 8 байт в 7

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>

void compress(char *input, char *output);
void decompress(char *input, char *output);

main(int argc, char *argv[])
{
     if(argc !=4) {
          cout << "Usage: input output compress/
decompress\n";
          exit(1);
     }

     if(toupper(*argv[3]) == 'C')
          compress(argv[1], argv[2]);
     else
```

```
        decompress(argv[1], argv[2]);
    return 0;
}

// Compress
void compress(char *input, char *output)
{
    char ch, ch2, done, t;
    ifstream in(input, ios::in | ios::binary);
    ofstream out(output, ios::out | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        exit(1);
    }

    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

    done = 0;
    do {
    /* Получим первый байт. Его значащие биты будут
    распределены по восьмым битам последующих семи байт
    */
    ch = in.get();
    ch = ch << 1; // сдвигаем восьмой бит
    // теперь распределим оставшиеся 7 бит по остальным 7
байтам
    for(t=0; t<7; ++t) {
        ch2 = in.get();
        if(in.eof()) {
            ch2 = 0;
            done = 1;
        }
        ch2 = ch2 & 127;
        ch2 = ch2 | ((ch<<t) &128);
        out.put((char) ch2);
    }
} while(!done && !in.eof());

in.close();
out.close();
}
```

```
void decompress(char *input, char *output)
{
     char ch, ch2, t;
     char s[7];
     ifstream in(input, ios::in | ios::binary);
     ofstream out(output, ios::out | ios::binary);

     if(!in) {
          cout << "Cannot open input file.\n";
          exit(1);
     }

     if(!out) {
          cout << "Cannot open output file.\n";
          exit(1);
     }

     do {
     ch = 0;
     // Восстанавливаем первый байт
     for(t=0; t<7; ++t) {
          ch2 = in.get();
          s[t] = ch2 & 127;
          ch2 = ch2 & 128;
          ch2 = ch2 >> (t + 1);
          ch = ch | ch2;
          }

          out.put((char) ch2); // записываем восстановленный
первый байт

     // Записываем остальные 7 байт

     for(t = 0; t < 7; ++t) out.put(s[t]);
} while(!done && !in.eof());

in.close();
out.close();
}
```

Код манипуляций с битами достаточно сложен, так как биты, составляющие восьмой байт, должны быть сдвинуты на свои места. В C++ оператор >> осуществляет сдвиг вправо, а оператор << — сдвиг влево. Эти операторы выполняются на побитовом уровне. Внимательно изучите логику этой программы, так как вы должны быть уверены в том, что понимаете работу функций **compress()** и **decompress()**.

Преобразование 4-байтового набора в 3-байтовый

Как уже упоминалось в предыдущем разделе, строчные и прописные буквы требуют всего 52 символов, которые могут быть представлены с помощью 6 бит. Таким образом, все строчные и прописные символы алфавита, а также наиболее распространенные знаки препинания могут быть представлены с помощью 6-битного слова. Однако, как уже упоминалось, представления алфавитных символов в кодах ASCII содержат значения большие, чем 63, что требует как минимум 7 бит. Однако, несмотря на эту проблему, сжать символы в 6-битные слова все же возможно. Для этого следует отказаться от кодовой таблицы ASCII. Вместо этого присвоим каждому символу новое 6-битное значение, которое будет представлять собой индекс в таблице символов. Индексы будут сохраняться при сжатии файла, причем 4 значения индекса будут сжиматься в 3-байтные пакеты. Таким образом будет достигнута степень сжатия в 25%. При распаковке файла индексы будут использоваться для получения стандартного ASCII-кода для каждого символа.

Ниже приведена программа, осуществляющая эту схему сжатия данных:

```
// Сжатие 4 символов в 3

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>

void compress(char *input, char *output);
void decompress(char *input, char *output);
char IndexToChar(char ch);
char CharToIndex(char i);

char letters[] =
     "abcdefghijklmnopqrstuvwxyz"
     "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
     ".,!;;\n\r\x1a";
     union packet_type {
          char chrs[4];
          unsigned long bits;
          } packet;
main(int argc, char *argv[])
{
     if(argc!=4) {
          cout << "Usage: input output compress/decompress\n";
```

```
        exit(1);
    }

    if(toupper(*argv[3]) == 'C')
        compress(argv[1], argv[2]);
    else
        decompress(argv[1], argv[2]);

    return 0;
}

// Compress
void compress(char *input, char *output)
{
    char ch;
    int i;
    ifstream in(input, ios::in | ios::binary);
    ofstream out(output, ios::out | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        exit(1);
    }

    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

do {
    packet.bits = 0L;

    ch = in.get();
    if(in.eof()) break;
    packet.bits = CharToIndex(ch);
    packet.bits <<= 6;
    ch = in.get();
    if(in.eof()) ch = ' '; // дополняем пробелами файл нечетной длины
    packet.bits |= CharToIndex(ch);
    packet.bits <<= 6;
    ch = in.get();
    if(in.eof()) ch = ' '; // дополняем пробелами файл нечетной длины
    packet.bits |= CharToIndex(ch);
    packet.bits <<= 6;
```

```
    ch = in.get();
    if(in.eof()) ch = ' '; // дополняем пробелами файл нечетной длины
    packet.bits |= CharToIndex(ch);
    // выходной пакет
    for(i=0; i<3; i++) out.put(packet.chrs[i]);

    } while(!in.eof());

in.close();
out.close();
}

// Decompress
void decompress(char *input, char *output)
{
    char ch, chrs[4];
    int i;
    ifstream in(input, ios::in | ios::binary);
    ofstream out(output, ios::out | ios::binary);

    if(!in) {
        cout << "Cannot open input file.\n";
        exit(1);
    }

    if(!out) {
        cout << "Cannot open output file.\n";
        exit(1);
    }

do {
    packet.bits = 0L;

    for(i=0; i<3; i++) packet.chrs[i] = in.get();

    if(in.eof()) break;

// маскируем все, кроме 6 младших бит
    ch = packet.bits & 63
    ch = IndexToChar(ch);
    chrs[3] = ch;
    packet.bits >>= 6;

// маскируем все, кроме 6 младших бит
    ch = packet.bits & 63
```

```
        ch = IndexToChar(ch);
        chrs[2] = ch;

        packet.bits >>= 6;

// маскируем все, кроме 6 младших бит
        ch = packet.bits & 63
        ch = IndexToChar(ch);
        chrs[1] = ch;

        packet.bits >>= 6;

// маскируем все, кроме 6 младших бит
        ch = packet.bits & 63
        ch = IndexToChar(ch);
        chrs[0] = ch;

        for(i=0; i<4; i++) out.put(chrs[i]);
        } while(!in.eof());

in.close();
out.close();
}

// Нахождение индекса по символу
char CharToIndex(char ch)
{
        int i;
        for(i=0; i<63; i++)
                if(ch == letters[i]) return i;

        return -1;
}

// Нахождение символа по индексу
char IndexToChar(char i)
{
        return letters[i];
}
```

Эта программа работает следующим образом. Каждый символ открытого тек ста преобразуется в индекс в массиве **letters** с помощью функции **CharToIndex()**. Поскольку ни один из индексов не может быть числом большим, чем 63, для представления индекса требуются только 6 бит. После считывания 4 символов индексы упаковываются в 3 младших байта переменной unsigned long integer. Так как C++ оперирует с 8-битными байтами, которые пред

ставляют собой наименьшую единицу встроенных типов данных, для осуществления этой процедуры требуется выполнить некоторые операции на уровне битов. Вкратце алгоритму можно дать следующее словесное описание:

```
for(каждые 4 индекса) {
    Копировать 6 младших бит из следующего индекса в младшее слово
    переменной unsigned long integer.
    Сдвинуть все биты влево на 6 позиций
}
```

После того, как каждый набор из 4 байт будет сжат в пакет из 3 байт, этот пакет будет записан в выходной файл. Для того, чтобы декодировать закодированный файл, процесс повторяется в обратном порядке. Сначала происходит считывание каждого пакета, затем извлекаются индексы, после чего происходит преобразование в символы с помощью функции **IndexToChar()**.

Рекомендации для самостоятельной разработки

Вы можете разработать собственные высокоэффективные алгоритмы шифрования, используя битовые манипуляции. Попробуйте реализовать некоторые из предложенных здесь идей:

- Создайте алгоритм на базе XOR, который использовал бы длинный ключ.
- На базе предыдущей идеи создайте процедуру шифрования на базе XOR, которая осуществляет пакетное кодирование файлов (единовременное кодирование пакета из нескольких байт). Выходной результат шифрования пакета используйте в качестве ключа для шифрования следующего пакета.
- Вот еще одна очень интересная идея: закодируйте два разных исходных файла в один и тот же зашифрованный выходной файл. Если этот файл расшифровывать одним способом, будет получен первый файл, а при декодировании вторым способом — второй файл.
- Попробуйте "спрятать" файл после того, как он будет зашифроан. Для этого разбейте содержимое шифрованного файла на части и распределите их по нескольким разным файлам. Очень хорошим способом является добавление кусков зашифрованного файла в конец области данных исполняемых файлов. После этого реконструировать файл будет очень сложно.

Что касается сжатия данных, то оно предлагает изобретательным программистам практически ничем не ограниченное поле деятельности. Для начала попробуйте реализовать собственную версию алгоритма Лемпел Зива. Описа

ние этого алгоритма можно найти во многих компьютерных книгах и журналах. Если вы серьезно увлеклись сжатием данных, вам будет полезна следующая книга: Mark Nelson, “The Data Compression Book”, 1991, M&T Publishing, Inc, Redwood City, California. В ней вы найдете подробные описания разнообразных методов сжатия данных, включая метод Лемпел-Зива, кодировку Хаффмана и т. д.

В любом случае попробуйте реализовать самостоятельно следующую идею по сжатию данных. Представьте себе большой текстовый файл. Скорее всего, он содержит множество повторяющихся слов типа “the”, “and”, “which” и тому подобное. Можно построить таблицу, состоящую из слов, встречающихся в файле. После того, как это будет сделано, таблицу и поток индексов можно записывать в новый файл, который будет представлять собой сжатую версию оригинала. Для того, чтобы реконструировать файл, используйте индексы и таблицу для записи распакованной версии.

Глава 8

Интерфейс с функциями языка ассемблера

В этой главе объясняется, как включить низкоуровневый ассемблерный код в ваши программы на C++. Если вы похожи на остальных программистов на C++, то попытки выжать из своей программы максимум производительности — это стиль вашей жизни, а не занятие, которому вы предаетесь время от времени. Первым шагом оптимизации производительности программы является внимательное изучение лежащих в ее основе алгоритмов. Наконец, наступит и такой момент, когда для реализации этих алгоритмов вам придется программировать в машинных командах. Именно на этом этапе и вступает в игру язык ассемблера. Благодаря его использованию вы получите полный контроль над исполнением вашей программы. Как вы увидите из дальнейшего изложения, способ включения процедур ассемблера в ваш код на C++ зависит от конкретного компилятора, хотя процесс, описанный в данной главе, применим к большинству компиляторов.

Существует два способа интегрировать модули в кодах ассемблера в ваши программы на C++. Вы можете разработать процедуру на ассемблере независимо от остальных частей программы, после чего связать ее с остальными модулями вашей программы на этапе редактирования связей. Как альтернативу, вы можете использовать inline-возможности языка C++ по работе с кодом на ассемблере. В этой главе обсуждаются оба этих метода.

Несколько слов предупреждения: эта глава НЕ УЧИТ программировать на ассемблере — подразумевается, что вы уже умеете это делать. Если вы не знаете ассемблера, не пытайтесь тестировать приведенные здесь примеры, так как при этом вы можете сделать незначительную ошибку, не заметить этого — и создать себе крупную неприятность типа уничтожения содержимого

всего вашего жесткого диска. Однако, если вы знакомы с программированием на ассемблере, вы не испытаете проблем с тестированием приведенных примеров и построением собственных функций на ассемблере.

Для чего нужно использование языка ассемблера?

Хотя язык C++ обладает богатыми возможностями, в ряде случаев вам может потребоваться написание функций на языке ассемблера. Ниже перечислены четыре основных причины, по которым может возникнуть такая потребность.

- Получение выигрыша в скорости и эффективности
- Выполнение аппаратно-зависимой функции, недоступной в C++
- Использование библиотек от сторонних поставщиков
- Выполнение действий в обход запретов C++ и операционной системы

Рассмотрим эти причины более подробно.

Хотя компиляторы C++, как правило, производят исключительно эффективный, быстрый и компактный объектный код, ни один компилятор не может стабильно создавать код, который в любом случае был бы лучше кода, написанного мастером программирования на ассемблере. По большей части эти незначительные различия не играют существенной роли, и затраты времени на написание кода на ассемблере не оправдывают себя. Однако, в некоторых случаях имеет смысл написание на ассемблере отдельных функций с целью повышения производительности. Например, можно написать на ассемблере математический пакет для выполнения операций с плавающей точкой, поскольку эти операции используются часто и оказывают большое влияние на скорость выполнения программ. В ряде случаев написание процедур на ассемблере уменьшает размер объектного кода. Кроме того, некоторые аппаратные устройства могут требовать точного временного квантования, а это значит, что для корректного обращения с таймером вам придется программировать на ассемблере.

Многие процессоры имеют команды, которые большинство компиляторов C++ выполнять не может. Например, при программировании для одного из процессоров семейства 8086 вы не сможете изменять сегменты данных с помощью стандартных инструкций C++. Кроме того, стандартные утверждения C++ не позволяют генерировать программных прерываний или управлять содержимым конкретных регистров.

В некоторых случаях вы захотите воспользоваться специализированными библиотеками, написанными на других языках, таких, как FORTRAN (или даже ассемблер). К примеру, может потребоваться использование специализированной библиотеки, которая управляет роботом или вычерчивает специаль

ные символы на плоттере. В некоторых случаях эти процедуры можно просто связать на этапе редактирования связей с кодом, откомпилированным с помощью вашего компилятора. Иногда может возникнуть необходимость в написании интерфейсного модуля, устраняющего различия в интерфейсе, используемом вашим компилятором и специализированными процедурами.

Наконец, есть еще одна причина, по которой может потребоваться язык ассемблера. Эта причина не столь благородна, как предыдущие. Обычно действия в обход ограничений С++ и операционной системы нельзя назвать хорошей идеей, но бывают и такие ситуации, когда это необходимо. Достигнуть этого можно с помощью языка ассемблера. Например, в отношении исходного кода на С++ действует ограничение по доступу к защищенным членам класса — этот доступ могут получить только другие члены того же класса. При использовании языка ассемблера защищенные члены так же доступны, как и открытые (public). Возможно, в обычных обстоятельствах (то есть, большую часть времени) вам и не нужно разрушать стену инкапсуляции. Однако, в исключительных случаях и это может понадобиться. К примеру, такой случай возможен после краха системы, когда вам во что бы то ни стало нужно восстановить ценные данные. Поскольку язык ассемблера предоставляет доступ к самым низкоуровневым машинным инструкциям, вы можете (в большей или меньшей степени) исполнять любые типы операций по своему выбору.

Какова бы ни была причина, заставившая вас включать в свою программу на С++ код на ассемблере, методы, описанные в этой главе, помогут вам в этом.

Основные принципы интерфейса с языком ассемблера

На интерфейс между С++ и ассемблером влияют четыре основных причины:

- Тип процессора
- Используемая модель памяти
- Соглашения о вызовах, установленные для компилятора
- Генерируемый код (16- или 32-разрядный)

Рассмотрим все эти факторы более подробно

Каждый процессор определяет собственный язык ассемблера. Таким образом, процессор, для которого разрабатывается код, оказывает самое серьезное влияние на написание модуля на языке ассемблера и его связывании с программой на С++.

Большинство процессоров позволяют по-разному организовать работу с памятью. Каждый из таких методов организации памяти называется *моделью памяти*. Модель памяти определяет такие характеристики, как размер указателя (то есть, адреса) и предельную величину самостоятельного объекта.

Известны следующие модели памяти: миниатюрная (tiny), малая (small), компактная (compact), средняя (medium), большая (large), громадная (huge) и плоская (flat). Спецификации этих моделей обсуждаются в следующем разделе.

Каждый компилятор С++ может определять собственные соглашения о вызовах, которые определяют правила, в соответствии с которыми функция получает и возвращает информацию. Для того, чтобы обеспечить взаимодействие кода на ассемблере с программой на С++, необходимо знать соглашения о вызовах, установленные для вашего компилятора.

Как, вероятно, вы уже знаете, более старые модели процессоров используют 16-разрядные регистры. Процессоры более современных моделей используют как 16-разрядные регистры (в целях совместимости), так и 32-разрядные регистры. Еще одним фактором, сильно осложняющим ситуацию, является то, что хотя новейшие процессоры и могут использовать 32-разрядные регистры, многим операционным системам (таким, как DOS и Windows 3.1) это недоступно. (Большей частью они оперируют с 16-разрядными значениями.) Инструкции ассемблерного кода для 16-разрядного режима часто существенно отличаются от аналогичных инструкций для 32-разрядного режима. Кроме того, при работе в 32-разрядном режиме, целые (integers) имеют длину 32 бита, а не 16, как было принято ранее.

Только по заданному количеству переменных, влияющим на взаимодействие кода на ассемблере с программой на С++, все возможные вариации предусмотреть просто невозможно. Поэтому необходимо сделать конкретные предположения относительно рабочей среды. Следовательно, мы должны сразу же оговорить все предположения, используемые в данной главе. Во-первых, большинство примеров используют 16-разрядный код, малую модель памяти, и требуют как минимум процессора 80286. Это означает, что примеры будут работать практически на любом компьютере. Однако, в число примеров включен и пример 32-разрядного кода, использующий плоскую модель памяти. (Эта программа предназначена для 32-разрядной операционной системы, например, Windows 95.) Для этого 32-рарядного примера требуется процессор 80386 или более мощный. Кроме того, здесь приведен еще один пример 16-разрядного кода, разработанный для гигантской (huge) модели памяти. Поскольку мы сделали предположение об использовании процессора семейства 8086, во всех примерах используется язык ассемблера, соответствующий процессорам семейства 8086. Примеры интерфейса языка ассемблера с программами на С++, рассматриваемые в данной главе, подразумевают использование соглашений о вызовах, соответствующих следующим компиляторам С++: Borland С++ (v. 4.5) и Microsoft С++ (v. 4). Несмотря на это, большинство из приведенной здесь информации применимо и к другим компиляторам С++. Даже в том случае, если у вас другой тип процессора или другой компилятор С++, нижеприведенная дискуссия может послужить вам в качестве путеводителя. Однако, при этом не следует забывать и о том, что интерфейс с языком ассемблера является одной из наиболее сложных тем, подходить к которой следует с осторожностью.

Соглашения о вызовах для компилятора С++

Соглашение о вызовах представляет собой метод, который каждый конкретный компилятор С++ использует для передачи информации функциям и возврата значений. Теоретически все компиляторы С++ для передачи аргументов функциям используют стек. Если аргумент принадлежит к одному из встроенных типов данных или является структурой, классом, объединением или перечислением, то в стек помещается фактическое значение. Если аргумент представляет собой массив, то в стек помещается его адрес. Когда функция С++ начинает выполняться, она извлекает из стека значение параметра. По завершении работы функции С++ она передает возвращаемое значение вызывающей процедуре. Как правило, эти значения возвращаются в регистрах, хотя некоторые типы значений (особенно большие) возвращаются через стек или (иногда) через внутреннюю глобальную переменную.

Соглашение о вызовах точно определяет, содержимое каких регистров в обязательном порядке должно сохраняться и какие регистры можно свободно использовать. Часто компилятор создает объектный код, которому требуется только часть доступных регистров конкретного процессора. Вы должны сохранять содержимое регистров, используемых компилятором. Как правило, это делается путем помещения их содержимого в стек на то время, пока регистры будут использоваться компилятором, с последующим извлечением значений по завершении их использования. Все остальные регистры вы можете использовать свободно.

При написании модуля на ассемблере, взаимодействующего с программой на языке С++, необходимо следовать соглашениям, определенным для вашего компилятора. Только таким образом вы можете обеспечить корректное взаимодействие процедур на ассемблере с кодом на С++. Следующий раздел подробно рассматривает соглашения о вызовах, принятые для компиляторов Microsoft и Borland С++ при работе в 16-разрядном режиме. Следует помнить, что соглашения о вызовах чувствительны к используемой модели памяти, а также учитывают тип генерируемого кода (16- или 32-разрядный).

Соглашения о вызовах для Microsoft/Borland С++

Ниже приведен краткий обзор соглашений о вызовах, используемых Microsoft С++ и Borland С++ при работе в 16-разрядном режиме. Как и в случае с большинством других компиляторов С++, и Borland С++, и Microsoft Visual С++ передают функциям аргументы через стек. Аргументы помещаются в стек, начиная с крайнего правого, и заканчивая крайним левым. Таким образом, если обрабатывается вызов функции:

```
func (a, b, c)
```

то первым аргументом, помещенным в стек, будет **c**, за ним последует **b**, и, наконец, **a**. Количество байт, занимаемое в стеке каждым из типов данных (в предположении использования 16-разрядного кода) приведено в таблице 8-1.

Таблица 8.1. Количество байт в стеке, необходимое для сохранения каждого из типов данных при передаче параметров функции с использованием компиляторов Borland и Microsoft C++ (16-разрядный компилятор)

Тип данных	Количество байт
char	2
short	2
signed char	2
signed short	2
unsigned char	2
unsigned short	2
int	2
signed int	2
unsigned int	2
long	4
unsigned long	4
float	4
double	8
long double	10
(near) pointer	2 (только смещение)
(far) pointer	4 (сегмент и смещение)

При входе в функцию на ассемблере содержимое регистра BP должно быть сохранено в стеке, после чего в регистр BP помещается текущее значение указателя стека (SP). Кроме того, вы должны сохранять и содержимое таких регистров, как SI, DI, CS, SS и DS (если ваша процедура использует их). Перед возвратом ваша функция на ассемблере должна восстановить значения регистров BP, SI, DI, CS, SS и DS и возвратить в исходное состояние указатель стека (SP). Поскольку соглашения о регистрах со временем могут изменяться, вы должны в обязательном порядке найти список регистров, значения которых должны сохраняться, в руководстве пользователя, относящемся к вашему компилятору.

Если ваша функция на языке ассемблера возвращает 8- или 16-битное значение, оно будет помещено в регистр AX. В противном случае это значение будет возвращено в соответствии с правилами, заданными в таблице 8-2.

Соглашения о вызовах для 32-разрядного режима будут похожи на эти же соглашения для 16-разрядного кода, за исключением того, что расширенные 32-разрядные регистры должны сохраняться.

Таблица 8.2. Использование регистров для возвращаемых значений (соглашение, принятое для компиляторов Microsoft/Borland C++)

Тип значения	Регистр(ы) и пояснение
char	AL
unsigned char	AL
short	AX
unsigned short	AX
int	AX
unsigned int	AX
long	младшее слово - AX; старшее слово - DX
unsigned long	младшее слово - AX; старшее слово - DX
float и double	Возвращается адрес значения, AX содержит смещение, DX - сегмент
struct и union	Возвращается адрес значения, AX содержит смещение, DX - сегмент
(near) pointer	AX
far) pointer	Смещение в AX, сегмент в DX

Одно последнее замечание: программа на C++ выделяет в стеке пространство для локальных данных. При написании функций на ассемблере вы должны использовать для локальных данных ту же процедуру.

Несколько слов о моделях памяти

Как уже упоминалось, модель памяти влияет на то, как должен быть написан модуль на ассемблере. Хотя подробное рассмотрение различных моделей памяти и режимов адресации, поддерживаемых процессорами семейства 8086, выходит за рамки материала, который может и должен рассматриваться в данной книге, мы дадим краткий обзор, которого должно быть достаточно для понимания материала, изложенного в этой главе.

При работе с сегментированной моделью памяти процессоры семейства 8086 просматривают память в виде набора фрагментов размером по 64 K. Каждый такой фрагмент называется сегментом. Адрес каждого байта памяти определяется адресом сегмента (хранящимся в сегментном регистре процессора) и его смещением в пределах сегмента (хранящимся в другом регистре). Как сегмент, так и смещение представляют собой 16-битные значения. При доступе к адресу памяти, лежащему в пределах текущего сегмента, для получения соответствующего байта требуется только 16-разрядное смещение. Однако, если искомый адрес памяти лежит за пределами текущего сегмента, потребуется загрузить оба 16 разрядных значения (сегмент и смещение). Таким образом, при доступе к

памяти, лежащей в пределах текущего сегмента, компилятор С++ может трактовать указатель, вызов или инструкцию перехода (jmp) как 16-битный объект. При доступе к памяти, расположенной за пределами текущего сегмента, компилятор должен рассматривать указатели, вызовы и инструкции перехода как 32-разрядные объекты.

Учитывая сегментированную природу памяти процессоров семейства 8086, можно организовать память в соответствии с одной из шести нижеследующих моделей (перечисленных в порядке возрастания времени исполнения кода):

Tiny	Все сегментные регистры установлены на одно значение, и вся адресация выполняется с использованием 16 бит. Это означает, что код, данные и стек должны разместиться в пределах одного 64-килобайтного сегмента. Наивысшая скорость исполнения программы.
Small	Весь код должен разместиться в пределах одного 64-килобайтного сегмента, все данные —в пределах другого 64-килобайтного сегмента. Все указатели имеют длину 16 бит. Программа исполняется почти так же быстро, как в предыдущей модели.
Medium	Все данные должны разместиться в пределах одного сегмента, в то время, как код может использовать несколько сегментов. Все указатели на данные имеют длину 16 бит, но все вызовы и переходы требуют 32-разрядных адресов. Быстрый доступ к данным, но медленное исполнение кода.
Compact	Весь код должен разместиться в пределах одного 64-килобайтного сегмента, а данные могут использовать несколько сегментов. Несмотря на это, ни одни элемент данных не может превышать 64K. Все указатели на данные — 32-разрядные, но переходы и вызовы могут использовать 16-разрядные адреса. Медленный доступ к данным, но более быстрое исполнение кода.
Large	Как код, так и данные используют несколько сегментов. Все указатели 32-разрядные. Однако, ни один элемент данных не может превышать 64K. Медленное исполнение программы.
Huge	Как код, так и данные используют несколько сегментов. Все указатели 32-разрядные. Отдельные элементы данных могут превышать 64K. Самое медленное исполнение.

Как можно было предположить, доступ к памяти через 16-битные указатели будет существенно быстрее, чем при использовании 32-битных указателей, поскольку при каждой ссылке на адрес памяти в процессор загружается ровно вполовину меньшее количество бит.

Более современные процессоры, такие, как 80486 и Pentium, могут работать еще с одной моделью памяти — плоской (flat) . При этом все адреса имеют длину 32 бита и не являются комбинацией сегмента и смещения. Плоская модель памяти используется при работе в 32-разрядном режиме. Таким образом,

современные процессоры к шести вышеперечисленным сегментированным моделям памяти добавили еще одну. Плоская модель памяти концептуально совершеннее и принципиально понятнее, поскольку принципы ее работы соответствуют интуитивному представлению большинства людей о работе памяти. Все адреса, от начала и до конца, представляют собой единые уникальные значения. Однако, вследствие необходимости обеспечения обратной совместимости 16-разрядные сегментированные модели все еще широко распространены. Поэтому, несмотря на то, что будущее принадлежит 32-разрядной плоской модели памяти, 16-разрядные сегментированные модели еще долго будут в ходу.

Для большинства примеров, приведенных в этой книге, будет достаточно малой (small) модели памяти. Эта модель является широко распространенной в качестве своего рода "нижней границы": практически каждый, кто имеет в своем распоряжении компьютер с процессором на базе 8086 и компилятор C++, может создать программу с малой моделью памяти.

Разработка функции на ассемблере

Простейшим способом научиться создавать функции на языке ассемблера, совместимые с соглашениями о вызовах вашего компилятора, является изучение способа, которым этот компилятор генерирует объектный код. Почти все компиляторы C++ имеют опцию компиляции (compile-time option), вызывающую вывод ассемблерного листинга кода, генерируемого в процессе компиляции. Изучив такой файл, вы не только многое узнаете об интерфейсе с компилятором, но и поймете, как он в действительности работает.

Для генерации вывода ассемблерного кода в процессе компиляции с использованием Borland C++, укажите опцию **–S** при использовании компилятора из командной строки. Для того, чтобы добиться того же самого при работе с компилятором Microsoft, следует выбрать соответствующую опцию в Developer Studio или указать опцию **—FA** при использовании компилятора из командной строки. Код на ассемблере будет помещен в файл с тем же именем, что и имя файла исходной программы на C++, но получит расширение .ASM. В этой главе приведены листинги кодов на ассемблере, призванные проиллюстрировать работу компилятора C++ при обработке различных типов операций.

Примечание: В некоторых листингах удалены строки, содержащие отладочную информацию. Это сделано для того, чтобы не затемнять смысл изложения. Удаление этих строк никак не скажется на работе ассемблерного кода.

Передача аргументов функции

Так как большинство функций на языке ассемблера будут оперировать с параметрами и возвращаемыми значениями, начнем с изучения простого примера, в котором программа передает два целых аргумента функции, которая затем возвращает значение. Эту задачу выполняет следующая короткая программа на C++:

```
int sum;
int add(int a, int b);

main()
{
     sum = add(10, 12);
     return 0;
}
add(int a, int b)
{
     int t;

     t = a + b;
     return t;
}
```

Переменная **sum** здесь намеренно декларируется как глобальная. Это сделано для того, чтобы продемонстрировать работу компилятора как с локальными, так и с глобальными данными. Если назвать эту короткую программу именем **test**, то компилятор Borland C++ создаст файл **test.asm** в результате выполнения следующей командной строки:

bcc -S test.cpp

Программа будет откомпилирована с использованием малой модели памяти. Содержимое файла **test.asm** приведено ниже:

```
          .286p
          ifndef    ??version
?debug    macro
          endm
publicdll macro     name
          public    name
          endm
$comm     macro     name,dist,size,count
          comm dist name:BYTE:cout*size
          endm
          else
$comm     macro     name,dist,size,count
```

```
        comm dist  name[size]:BYTE:count
        endm
        endif
        ?debug    V  301h
        ?debug    S  "test.cpp"
        ?debug    C  E98FB09E1F08746573742E637070
_TEXT   segment   byte public 'CODE'
_TEXT   ends
DGROUP  group     _DATA,_BSS
        assume    cs:_TEXT,ds:DGROUP
_DATA   segment   word public 'DATA'
d@      label     byte
d@w     label     word
_DATA   ends
_BSS    segment   word public 'BSS'
b@      label     byte
b@w     label     word
_sum    label     word
        db   2 dup (?)
_BSS    ends
_TEXT   segment   byte public 'CODE'
   ;
   ;    main()
   ;
        assume    cs:_TEXT,ds:DGROUP
_main   proc near
        push bp
        mov  bp,sp
   ;
   ;    {
   ;       sum = add(10,12);
   ;
        push 12
        push 10
        call near ptr @add$qii
        add  sp,4
        mov  word ptr DGROUP:_sum,ax
   ;
   ;
   ;    return 0;
   ;
        xor  ax,ax
        pop  bp
        ret
```

```
      ;
      ;    }
      ;
           pop  bp
           ret
_main      endp
      ;
      ;    add(int a,int b)
      ;
           assume    cs:_TEXT,ds:DGROUP
@add$qii             proc near
           enter     2,0
      ;
      ;    {
      ;    int t;
      ;
      ;    t = a + b;
      ;
           mov  ax,word ptr [bp+4]
           add  ax,word ptr [bp+6]
           mov  word ptr [bp-2],ax
      ;
      ;    return t;
      ;
           mov  ax,word ptr [bp-2]
           leave
           ret
      ;
      ;    }
      ;
           leave
           ret
@add$qii        endp
           ?debug    C E9
           ?debug    C FA00000000
_TEXT      ends
_DATA      segment   word public 'DATA'
s@         label     byte
_DATA      ends
_TEXT      segment   word public 'CODE'
_TEXT      ends
_s@        equ  s@
           public    _sum
           public    @add$dii
           public    _main
           end
```

Давайте внимательно изучим эту версию программы в кодах ассемблера. Первым фактом, на который следует обратить внимание, является то, что компилятор автоматически добавляет символ подчеркивания перед именами **sum** и **main**. Это делается для того, чтобы избежать путаницы с внутренними именами компилятора. Фактически символ подчеркивания добавляется перед всеми именами глобальных переменных. (Это обычная практика, используемая большинством компиляторов.) Далее, обратите внимание на то, как изменилось имя функции **add()**. Поскольку C++ поддерживает перегрузку функций, компилятор должен сконструировать уникальное имя функции для каждой ее перегруженной версии. Компилятор выполняет это с помощью процедуры, известной под названием "смешивания имен" (*name mangling*). При этом информация о параметрах функции кодируется внутри ее внутреннего имени. Хотя эта программа и не перегружает функцию **add()**, компилятор все равно действует в соответствии с этим соглашением.

Теперь рассмотрим код, ассоциированный с **_main**. Первое, что делает этот код — помещает в стек содержимое регистра BP, после чего помещает значение SP в BP. Далее, два аргумента функции **add** помещаются в стек, и вызывается функция **add**. После выхода из функции **add** стек переустанавливается (инструкцией **add sp,4**). После этого возвращаемое значение, которое находится в регистре AX, перемещается в **_sum**. Наконец, **_main** восстанавливает BP и возвращает 0.

Выполнение функции **add** начинается с инструкции ENTER, которая строит стандартный фрейм стека. Эта инструкция сохраняет в стеке содержимое BP, перемещает содержимое SP в BP и выделяет в стеке пространство для локальных переменных. После того, как это будет выполнено, можно получить доступ к параметрам, используя положительные смещения относительно BP. Доступ к локальным переменным можно получить, используя отрицательное смещение относительно BP. Следующие три строки кода выполняют сложение чисел и помещают сумму в стек в позицию переменной **t**. Обратите внимание на то, как функция **add** получает доступ к параметрам путем индексации BP. После того, как сложение будет выполнено, возвращаемое значение (в данном случае **t**), загружается в AX. После этого вызывается инструкция LEAVE, которая отменяет операции, выполненные инструкцией ENTER, после чего функция возвращает управление.

Для того, чтобы этот файл можно было использовать, его необходимо сначала ассемблировать, а затем обработать редактором связей, входящим в состав run-time пакета, поставляющегося с компилятором C++. Например, если вы работаете с компилятором BORLAND и имеете в своем распоряжении TASM (ассемблер BORLAND), то для ассемблирования и редактирования связей этого файла необходимо дать следующую команду:

```
bcc test.asm
```

Borland C++ автоматически вызывает TASM, который выполняет ассемблирование файла. Затем Borland C++ автоматически выполняет редактирование связей с помощью своей библиотеки времени выполнения (run-time library).

Как уже упоминалось, следует иметь в виду, что разные компиляторы генерируют код, имеющий некоторые различия. Далее, на генерируемый ассемблерный код оказывают существенное влияние используемая модель памяти а также тип генерируемого кода (16- или 32-разрядный). Для того, чтобы почувствовать эту разницу, рассмотрим еще одну программу на ассемблере. Этот код был получен в результате компиляции той же самой программы на C++ с помощью 32-разрядной версии компилятора Microsoft Visual C++ 4, с использованием плоской модели памяти (с 32-разрядной адресацией). Обратите внимание на сходство и различия с кодом, порожденным 16-разрядным компилятором Borland.

```
         TITLE    test.cpp
         .386P
include  listing.inc
if @Version gt 510
.model FLAT
else
_TEXT    SEGMENT PARA USE32 PUBLIC 'CODE'
_TEXT    ENDS
_DATA    SEGMENT DWORD USE32 PUBLIC 'DATA'
_DATA    ENDS
CONST    SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST    ENDS
_BSS     SEGMENT DWORD USE32 PUBLIC 'BSS'
_BSS     ENDS
_TLS     SEGMENT DWORD USE32 PUBLIC 'TLS'
_TLS     ENDS
FLAT     GROUP _DATA, CONST, _BSS
         ASSUME CS: FLAT, DS: FLAT, SS: FLAT
endif
_BSS     SEGMENT
?sum@@3HA DD   01H DUP (?)                          ;sum
_BSS     ENDS
PUBLIC   ?add@@YAHHH@Z                          ;add
PUBLIC   _main
_TEXT    SEGMENT
; File test.cpp
_main    PROC NEAR
; Line 5
         push ebp
         mov  ebp,esp
         push ebx
         push esi
         push edi
```

```
; Line 6
          push 12                                    ;0000000cH
          push 10                                    ;0000000aH
          call ?add@@YAHHH@Z                         ;add
          add  esp,8
          mov  DWORD PTR ?sum@@3HA,eax               ;sum
; Line 8
          xor  eax,eax
          jmp  $L173
;Line 9
$L173:
          pop  edi
          pop  esi
          pop  ebx
          leave
          ret  0
_main     ENDP
_a$ = 8
_b$ = 12
_t$ = -4
?add@@YAHHH@Z PROC NEAR                              ;add
;Line 12
          push ebp
          mov  ebp,esp
          sub  esp,4
          push ebx
          push esi
          push edi
;Line 15
          mov  eax, DWORD PTR _b$[ebp]
          add  eax, DWORD PTR _a$[ebp]
          mov  DWORD PTR _t$[ebp], eax
;Line 16
          mov  eax, DWORD PTR _t$[ebp]
          jmp  $L176
; Line 17
$L176:
          pop  edi
          pop  esi
          pop  ebx
          leave
          ret  0
?add@@YAHHH@Z ENDP                                   ;add
_TEXT     ENDS
END
```

Единственным существенным различием между этой версией программы и версией, порожденной Borland C++, является использование 32-разрядных инструкций. Например, вместо сохранения 16-разрядного регистра BP 32-разрядная версия сохраняет EBP, 32-разрядную версию BP. Далее, все ссылки теперь делаются только на 32-разрядные регистры. Обратите внимание на то, что версия Microsoft явным образом создает фрейм стека для **add** (вместо использования инструкции ENTER). Однако, она продолжает использовать инструкцию LEAVE для очистки стека перед возвращением управления. Кроме того, как вы можете видеть из этого листинга, Microsoft использует кодирование имен для **add** и делает это по той же причине, что и Borland. Имейте в виду, что все компиляторы С++ в той или иной форме используют кодирование имен функций, чтобы обеспечить уникальность имен для перегруженных функций.

Вызов библиотечных функций и операторов

Как правило, стандартные библиотечные функции и функции-операторы (такие, как операторы ввода-вывода) вызываются из ассемблерного кода точно таким же способом, как вызывается написанная вами функция. Все аргументы помещаются в стек, выполняется инструкция CALL, после чего производится очистка стека. Приведенный ниже пример вызывает функцию **ostream operator<<( )** и библиотечные функции **srand( )** и **rand( )**:

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

int sum;
int randnum;

int add(int a, int b);

main()
{
     sum = add(10, 12);
     cout << sum;

     srand(1000);
     randnum = rand();
     cout << randnum;

     return 0;
     }
```

```
add(int a, int b)
{
     int t;

     t = a + b;
     return t;
}
```

Эта программа порождает нижеследующий код на ассемблере. Обратите особое внимание на то, каким образом вызываются оператор вывода и библиотечные функции.

```
          .286p
          ifndef    ??version
?debug    macro
          endm
publicdll macro     name
          public    name
          endm
$comm     macro     name,dist,size,count
          comm dist name:BYTE:cout*size
          endm
          else
$comm     macro     name,dist,size,count
          comm dist name[size]:BYTE:count
          endm
          endif
          ?debug    V  301h
          ?debug    S  "test.cpp"
_TEXT     segment   byte public 'CODE'
_TEXT     ends
DGROUP    group     _DATA,_BSS
          assume    cs:_TEXT,ds:DGROUP
_DATA     segment   word public 'DATA'
d@        label     byte
d@w       label     word
_DATA     ends
_BSS      segment   word public 'BSS'
b@        label     byte
b@w       label     word
 sum      label     word
          db   2 dup (?)
 randnum  label     word
          db   2 dup (?)
 BSS      ends
```

```
_TEXT     segment   byte public 'CODE'
     ;
     ;    main()
     ;
          assume    cs:_TEXT,ds:DGROUP
_main     proc near
     enter     4,0
     ;
     ;    {
     ;       sum = add(10,12);
     ;
          push 12
          push 10
          call near ptr @add$qii
          add  sp,4
          mov  word ptr DGROUP:_sum,ax
     ;
     ;    cout << sum;
     ;
          mov  ax,word ptr DGROUP:_sum
          mov  word ptr[bp-2],ax
          mov  ax, word ptr [bp-2]
          cwd
          push dx
          push ax
          push offset DGROUP:_cout
          call near ptr @ostream@$blsh$ql
     .    add  sp,6
     ;
     ;
     ;    srand(1000);
     ;
          push 1000
          call near ptr _srand
     pop  cx
     ;
     ;    randnum = rand();
     ;
          call near ptr _rand
          mov  word ptr DGROUP:_randnum,ax
     ;
     ;    cout << randnum;
     ;
          mov  ax,word ptr DGROUP:_randnum
          mov  word ptr[bp-4],ax
          mov  ax,word ptr [bp-4]
```

```
        cwd
        push dx
        push ax
        push offset DGROUP:_cout
        call near ptr @ostream@$blsh$ql
        add  sp,6
   ;
   ;
   ;    return 0;
   ;
        xor  ax,ax
        leave
        ret
   ;
   ;    }
   ;
        leave
        ret
_main   endp
   ;
   ;    add(int a,int b)
   ;
        assume    cs:_TEXT,ds:DGROUP
@add$qii      proc near
        enter     2,0
   ;
   ;    {
   ;    int t;
   ;
   ;    t = a + b;
   ;
        mov  ax,word ptr [bp+4]
        add  ax,word ptr [bp+6]
        mov  word ptr [bp-2],ax
   ;
   ;    return t;
   ;
        mov  ax,word ptr [bp-2]
        leave
        ret
   ;
   ;    }
   ;
        leave
        ret
```

```
@add$qii        endp
        ?debug   C E9
        ?debug   C FA00000000
_TEXT   ends
_DATA   segment  word public 'DATA'
s@      label    byte
_DATA   ends
_TEXT   segment  word public 'CODE'
_TEXT   ends
_s@     equ  s@
        extrn    @ostream@$blsh$ql:near
        extrn    _cout:word
_abs    equ  abs
_atoi   equ  atoi
        extrn    _rand:near
        extrn    _srand:near
        public   _sum
        public   _randnum
        public   @add$dii
        public   _main
        end
```

Обратите внимание на то, что к вызовам функции **ostream operator<<( )** применяется кодирование имен. Этого следовало ожидать, так как оператор-функция вывода **ostream** перегружается несколько раз (по одному разу для каждого из встроенных типов). При этом к функциям **rand( )** и **srand( )** этот тип кодирования не применяется. Фактически, если не обращать внимания на добавление символа подчеркивания, их имена не изменились. Причина этого проста: **rand( )** и **srand( )** являются частью стандартной библиотеки С (которая, в свою очередь, является составной частью С++), а С не поддерживает перегрузки функций. Это означает, что ни одна из частей стандартной библиотеки С не перегружается. Таким образом, нет необходимости и в кодировании имен функций. Тем не менее, компилятор добавляет в начало обоих имен символ подчеркивания, что является стандартной практикой при компиляции функций С.

В целом можно сказать, что если ваша программа на ассемблере должна вызывать функцию из стандартной библиотеки С, то к имени этой функции необходимо добавлять начальный символ подчеркивания. Для того, чтобы вызвать перегруженную функцию или оператор С++, вы должны употребить для них правильную форму закодированного имени. Простейшим способом получить такое имя является написание простой программы на С++, которая бы вызывала эту же функцию таким же способом, после чего откомпилировать эту программу с построением листинга кода на ассемблере. В этом листинге вы с легкостью найдете нужное вам имя.

Получение доступа к структурам и классам

Хотя они несколько более сложны, чем рассмотренные ранее библиотечные функции, члены структур и классов представляют не намного больше трудностей при использовании для интерфейса с кодом на ассемблере. Ниже приведен пример короткой программы, которая использует структуру.

```
#include <iostream.h>

struct mystruct {
     int i;
     int j
     };

main()
{
     mystruct ob;

     ob.i = 10;
     ob.j = 99;

     cout << ob.i << " " << ob.j << endl;

     return 0;
}
```

Ассемблерный код, сгенерированный на этапе компиляции, выглядит следующим образом:

```
         .286p
         ifndef    ??version
?debug   macro
         endm
publicdll macro    name
         public    name
         endm
$comm    macro     name,dist,size,count
         comm dist name:BYTE:cout*size
         endm
         else
$comm    macro     name,dist,size,count
         comm dist name[size]:BYTE:count
         endm
```

```
            endif
            ?debug    V  301h
            ?debug    S  "test.cpp"
_TEXT       segment   byte public 'CODE'
_TEXT       ends
DGROUP      group     _DATA,_BSS
            assume    cs:_TEXT,ds:DGROUP
_DATA       segment   word public 'DATA'
d@          label     byte
d@w         label     word
_DATA       ends
_BSS        segment   word public 'BSS'
b@          label     byte
b@w         label     word
_BSS        ends
_TEXT       segment   byte public 'CODE'
     ;
     ;      main()
     ;
            assume    cs:_TEXT,ds:DGROUP
_main       proc near
            enter     10,0
            push si
     ;
     ;      {
     ;      mystruct ob;
     ;
     ;      ob.i = 10;
     ;
            mov  word ptr [bp-4],10
     ;
     ;      ob.j = 99
     ;
            mov  word ptr [bp-2],99
     ;
     ;
     ;      cout <<ob.i << " " << ob.j << endl;
     ;
            mov  ax,word ptr [bp-2]
            mov  word ptr [bp-6], ax
            mov  ax, word ptr [bp-4]
            mov  word ptr [bp-8], ax
            mov  ax, word ptr [bp-8]
            cwd
            push dx
```

```
        push ax
        push offset DGROUP:_cout
        call near ptr @ostream@$blsh$ql
        add  sp, 6
        mov  si, ax
        push 0
        push offset DGROUP:s@
        push si
        call near ptr @ostream@outstr$qpxctl
        add  sp,6
        mov  ax,word ptr[bp-6]
        cwd
        push ds
        push ax
        push si
        call near ptr @ostream@$blsh@ql
        add  sp,6
        mov  word ptr[bp-10],ax
        push word ptr[bp-10]
        call near ptr @endl$qr7ostream
        pop  cx
   ;
   ;
   ;      return 0;
   ;
        xor  ax,ax
        jmp  short @1@86
@1@86:
   ;
   ;    }
   ;
        pop  si
        leave
        ret
_main   endp
        ?debug    C E9
        ?debug    C FA00000000
_TEXT   ends
_DATA   segment   word public 'DATA'
s@      label     byte
        db   ' '
        db   0
_DATA   ends
_TEXT   segment   byte public 'CODE'
_TEXT   ends
```

```
_s@  equ  s@
     extrn    @ostream@$blsh$ql:near
     extrn    @ostream@outstr$qpxctl:near
     extrn    _cout:word
     extrn    @endl$qr7ostream:near
     public   _main
     end
```

Как видите, доступ к членам структуры осуществляется точно так же, как к переменным других типов. Не забывайте, что структуры являются только логическими группировками, применимыми к коду высокого уровня на C++. На машинном уровне все данные являются одинаковыми. Поэтому в ассемблерной версии кода программы переменные-члены **i** и **j** сохраняются в стеке так же, как и любые другие локальные переменные.

Классы в этом смысле абсолютно аналогичны структурам. Рассмотрим, например, следующую программу:

```
#include <iostream.h>
class myclass {
     int i, j;
public:
     myclass(int x, int y) {i = x; j = y;}
     int geti() { return  i*2;}
     int getj() { return  j+3;}
     };

main()
{
     myclass ob(10, 20);

     cout << ob.geti() << " " << ob.getj() << endl;

     return 0;
}
```

Компиляция этой программы породит следующий файл на языке ассемблера:

```
          .286p
          ifndef   ??version
?debug    macro
          endm
publicdll macro    name
          public   name
          endm
```

```
$comm    macro     name,dist,size,count
         comm dist name:BYTE:cout*size
         endm
         else
$comm    macro     name,dist,size,count
         comm dist name[size]:BYTE:count
         endm
         endif
         ?debug    V  301h
         ?debug    S  "test.cpp"
_TEXT    segment   byte public 'CODE'
_TEXT    ends
DGROUP   group     _DATA,_BSS
         assume    cs:_TEXT,ds:DGROUP
_DATA    segment   word public 'DATA'
d@       label     byte
d@w      label     word
_DATA    ends
_BSS     segment   word public 'BSS'
b@       label     byte
b@w      label     word
_BSS     ends
_TEXT    segment   byte public 'CODE'
    ;
    ;    main()
    ;
         assume    cs:_TEXT,ds:DGROUP
_main    proc near
         enter     10,0
         push si
    ;
    ;    {
    ;       myclass ob(10,20);
    ;
         mov  word ptr [bp-4],10
         mov  word ptr [bp-2],20
    ;
    ;
    ;    cout << ob.geti() << " " << ob.getj() << endl;
    ;
         mov  ax,word ptr [bp-2]
         add  ax,3
         mov  word ptr [bp-6],ax
         mov  ax,word ptr [bp-4]
         add  ax,ax
```

```
        mov  word ptr [bp-8],ax
        mov  ax,word ptr [bp-8]
        cwd
        push dx
        push ax
        push offset DGROUP:_cout
        call near ptr @ostream@$blsh$ql
        add  sp,6
        mov  si,ax
        push 0
        push offset DGROUP:s@
        push si
        call near ptr @ostream@outstr$qpxctl
        sdd  sp,6
        mov  ax,word ptr [bp-6]
        cwd
        push dx
        push ax
        push si
        call near ptr @ostream@$blsh$ql
        add  sp,6
        mov  word ptr [bp-10],ax
        push word ptr [bp-10]
        call near ptr @endl$qr7ostream
        pop  cx
    ;
    ;
    ;   return 0;
    ;
        xor  ax,ax
        jmp  short @1@86
@1@86:
    ;
    ;   }
    ;
        pop  si
        leave
        ret
_main   endp
        ?debug    C E9
        ?debug    C FA00000000
_TEXT   ends
_DATA   segment   word public 'DATA'
s@      label     byte
        db   ' '
        db   0
```

```
_DATA    ends
_TEXT    segment  byte public 'CODE'
_TEXT    ends
_s@      equ  s@
         extrn    @ostream@$blsh$ql:near
         extrn    @ostream@outstr$qpxctl:near
         extrn    _cout:word
         extrn    @endl$qr7ostream:near
         public   _main
         end
```

С первого взгляда вы можете не заметить ничего необычного в этой программе на ассемблере. Однако, посмотрите еще раз внимательно на программу С++, которая использовалась для генерации этого кода. В особенности обратите внимание на то, что переменные **i** и **j** являются защищенными членами класса **myclass**. Очень важно понять, что такие понятия С++, как public и private, не транслируются в код на ассемблере. Любая часть программы на ассемблере может получить доступ к любому члену класса (открытому или защищенному). К примеру, обратите особое внимание на две строки кода, которые инициализируют защищенные переменные-члены объекта **ob i** и **j**:

```
mov  word ptr [bp-4],10
mov  word ptr [bp-2],20
```

Как вы можете видеть, доступ к переменным **ob.i** и **ob.j** осуществляется точно так же, как и к переменным любого другого типа. То, что они являются защищенными членами класса **myclass**, не оказывает никакого влияния на ассемблерный код, генерируемый компилятором. Подводя итоги, можно сказать, что инкапсуляция является просто логической конструкцией, имеющей значение только для исходного кода на С++. К функциям кода на ассемблере это понятие не имеет никакого отношения.

Тот факт, что защищенные члены класса не являются защищенными в программе на языке ассемблера, приводит к очевидному выводу: механизм инкапсуляции С++ можно обойти. Например, в только что рассмотренной программе переменным **ob.i** и **ob.j** можно присвоить новые значения путем простого добавления двух новых инструкций **mov**. Разумеется, обход инкапсуляции с помощью кода на ассемблере несет угрозу одному из самых важных (практически фундаментальных) свойств С++. Поэтому за исключением экстраординарных ситуаций не следует использовать функции ассемблера для этой цели.

У этой программы есть еще одна интересная особенность. Внимательно посмотрите на код, порожденный вызовами **geti( )** и **getj( )**. Поскольку эти функции члены определены в пределах класса **myclass**, они автоматически

получают свойства **inline**. Поэтому их код не вызывается, а генерируется **inline**. Именно по этой причине в ассемблерной версии программы вы не найдете инструкций **call**, вызывающих эти функции.

Использование указателей и ссылок

Когда указатели или ссылки используются в качестве аргументов функции, то функции передаются адреса, а не значения. Для того, чтобы понять, как это влияет на ассемблерную версию кода функции, рассмотрим следующую программу. В этой программе функция **get_val()** вызывается с использованием адреса **num**. На примере этой программы мы и проиллюстрируем код ассемблера, генерируемый при передаче указателей.

```
#include <iostream.h>

void get_val(int *ptr);

main()
{
     int num;

     get_val(&num);
     cout << num;

     return 0;
}

void get_val(int *ptr)
{
     *ptr = 100;
}
```

Версия этой программы на ассемблере выглядит следующим образом:

```
         .286p
         ifndef    ??version
?debug   macro
         endm
publicdll macro    name
         public    name
         endm
$comm    macro     name,dist,size,count
         comm dist name:BYTE:cout*size
         endm
```

```
        else
$comm   macro      name,dist,size,count
        comm dist name[size]:BYTE:count
        endm
        endif
        ?debug     V  301h
        ?debug     S  "test.cpp"
_TEXT   segment    byte public 'CODE'
_TEXT   ends
DGROUP  group      _DATA,_BSS
        assume     cs:_TEXT,ds:DGROUP
_DATA   segment    word public 'DATA'
d@      label      byte
d@w     label      word
_DATA   ends
_BSS    segment    word public 'BSS'
b@      label      byte
b@w     label      word
_BSS    ends
_TEXT   segment    byte public 'CODE'
    ;
    ;   main()
    ;
        assume     cs:_TEXT,ds:DGROUP
_main   proc near
        enter      4,0
    ;
    ;   {
    ;   int num;  .
    ;
    ;
    ;   get_val(&num);
    ;
        lea  ax,word ptr [bp-2]
        push ax
        call near ptr @get_val$qpi
        pop  cx
    ;
    ;   cout << num;
    ;
        mov  ax, word ptr [bp-2]
        mov  word ptr [bp-4],ax
        mov  ax, word ptr [bp-4]
        cwd
        push dx
```

```
        push ax
        push offset DGROUP:_cout
        call near ptr @ostream@$blsh$ql
        add  sp,6
    ;
    ;
    ;   return 0;
    ;
        xor ax,ax
        leave
    ret
    ;
    ;   }
    ;
        leave
        ret

_main   endp
    ;
    ;   void get_val(int *ptr)
    ;
        assume   cs:_TEXT, ds:DGROUP
@get_val$qpi  proc near
        push bp
        mov  bp,sp
        push si
        mov  si,word ptr [bp+4]
    ;
    ;   {
    ;   *ptr = 100;
    ;
        mov  word ptr [si], 100
    ;
    ;   }
    ;
        pop  si
        pop  bp
        ret
@get_val$qpi  endp
        ?debug   C E9
        ?debug   C FA00000000
_TEXT   ends
_DATA   segment word public 'DATA'
@s      label byte
_DATA   ends
```

```
_TEXT    segment byte public 'CODE'
_TEXT    ends
_s@      equ  s@
         extrn    @ostream@$blsh$ql:near
         extrn    _cout:word
         public   @get_val$qpi
         public   _main
         end
```

Как вы можете видеть, **get_val** вызывается с адресом **num**. Адрес **num** находится с помощью инструкции языка ассемблера LEA (Load Effective Address). В пределах функции **get_val** этот адрес загружает в **num** значение **100**, используя режим непрямой адресации семейства 8086.

Поскольку ссылочные параметры представляют собой (более или менее) автоматизированные указатели в C++, вы можете ожидать, что изменение вышеприведенной программы на C++ таким образом, чтобы функция **get_val()** использовала ссылку вместо указателя, не окажет большого влияния на код ассемблера, порожденный компилятором. И вы будете правы! Например, рассмотрим версию вышеприведенной программы, которая в качестве параметра использует ссылку:

```
#include <iostream.h>

void get_val(int &ref);

main()
{
    int num;

    get_val(num);
    cout << num;

    return 0;
}

void get_val(int &ref)
{
    ref = 100;
}
```

Ниже приведены соответствующие части этой программы в кодах ассемблера.

```
TEXT     segment  byte public 'CODE'
    ;
```

```
      ;     main()
      ;
            assume    cs:_TEXT,ds:DGROUP
_main       proc near
            enter     4,0
      ;
      ;     {
      ;     int num;
      ;
      ;
      ;     get_val(num);
      ;
            lea  ax,word ptr [bp-2]
            push ax
            call near ptr @get_val$qpi
            pop  cx
      ;
      ;     cout << num;
      ;
            mov  ax, word ptr [bp-2]
            mov  word ptr [bp-4],ax
            mov  ax, word ptr [bp-4]
            cwd
            push dx
            push ax
            push offset DGROUP:_cout
            call near ptr @ostream@$blsh$ql
            add  sp,6
      ;
      ;
      ;     return 0;
      ;
            xor ax,ax
            leave
            ret
      ;
      ;     }
      ;
            leave
            ret

_main       endp
      ;
      ;     void get_val(int &ref)
      ;
```

```
        assume   cs:_TEXT, ds:DGROUP
@get_val$qpi  proc near
        push bp
        mov  bp,sp
        push si
        mov  si,word ptr [bp+4]
    ;
    ;   {
    ;   ref = 100;
    ;
        mov  word ptr [si], 100
    ;
    ;   }
    ;
        pop  si
        pop  bp
        ret
@get_val$qpi  endp
        ?debug   C E9
        ?debug   C FA00000000
_TEXT   ends
_DATA   segment word public 'DATA'
@s      label byte
_DATA   ends
_TEXT   segment byte public 'CODE'
_TEXT   ends
_s@     equ  s@
        extrn    @ostream@$blsh$ql:near
        extrn    _cout:word
        public   @get_val$qpi
        public   _main
        end
```

Как вы можете видеть, код остается практически идентичным.

Пример, использующий гигантскую (huge) модель памяти

Как уже упоминалось, при использовании 16-разрядных компиляторов используемая модель памяти будет иметь глубокое влияние на способ написания ассемблерной программы, которая должна взаимодействовать с кодом, производимым вашей программой. Для того, чтобы наглядно проиллюстрировать это влияние, попробуйте откомпилировать программу на C++, приве-

денную в предыдущем разделе, используя гигантскую модель памяти. При использовании этой модели все вызовы библиотечных функций и глобальных данных будут дальними (FAR). Фактически это означает, что все вызовы будут требовать указания сегмента и смещения. Для того, чтобы выполнить это с помощью компилятора Borland, необходимо задать следующие опции компилятора **-mh**. В результате будет получен следующий код на ассемблере:

```
          .286p
          ifndef    ??version
?debug    macro
          endm
publicdll macro     name
          public    name
          endm
$comm     macro     name,dist,size,count
          comm dist name:BYTE:cout*size
          endm
          else
$comm     macro     name,dist,size,count
          comm dist name[size]:BYTE:count
          endm
          endif
          ?debug    V  301h
          ?debug    S  "test.cpp"
T2_TEXT   segment   byte public 'CODE'
T2_TEXT   ends
          assume    cs:T2_TEXT,ds:T2_DATA
T2_DATA   segment   word public 'FAR_DATA'
d@        label     byte
d@w       label     word
b@        label     byte
b@w       label     word
T2_DATA   ends
T2_TEXT   segment   byte public 'CODE'
     ;
     ;    main()
     ;
          assume    cs:T2_TEXT,ds:T2_DATA
_main     proc far
          enter     4,0
          push ds
          mov  ax, t2_DATA
          mov  ds,ax
     ;
     ;    {
```

```
;    int num;
;
;    get_val(num);
;
     push ss
     lea  ax,word ptr [bp-2]
     push ax
     call far ptr @get_val$qmi
     add  sp, 4
;
;    cout << num;
;
     mov  ax, word ptr [bp-2]
     mov  word ptr [bp-4],ax
     mov  ax, word ptr [bp-4]
     cwd
     push dx
     push ax
     push seg_cout
     push offset _cout
     call far ptr @ostream@$blsh$ql
     add  sp,8
;
;
;    return 0;
;
     xor ax,ax
     pop ds
     leave
     ret
;
;    }
;
     pop ds
     leave
     ret

_main      endp
;
;    void get_val(int &ref)
;
     assume   cs:T2_TEXT, ds:T2_DATA
@get_val$qmi  proc far
     push bp
     mov  bp,sp
;
;    {
;    ref = 100;
```

```
	;
		les  bx,dword ptr [bp+6]
		mov  word ptr es:[bx],100
	;
	;	}
	;
		pop  bp
		ret
@get_val$qmi  endp
		?debug   C E9
		?debug   C FA00000000
T2_TEXT	ends
T2_DATA	segment word public 'FAR_DATA'
@s		label byte
T2_DATA	ends
T2_TEXT	segment byte public 'CODE'
T2_TEXT	ends
_s@		equ  s@
		extrn    @ostream@$blsh$ql:far
		extrn    _cout:word
		public   @get_val$qmi
		public   _main
		end
```

Обратите внимание на два существенных различия между этой и предыдущей версиями. Во-первых, адрес **num** теперь требует 4 (а не 2) байт для хранения в стеке перед вызовом функции **get_val**. Это позволяет передать как сегмент (SS), так и смещение, характеризующие **num**. В пределах **get_val** доступ к **num** осуществляется по полному 32-разрядному адресу. Во-вторых, вызовы функций **get_val** и **ostream operator<<( )** теперь используют дальние (FAR) указатели. Если вы хотите связать свои функции, написанные на языке ассемблера, с кодом на C++, скомпилированном для модели **huge**, вы должны построить совместимый с этой моделью код возврата после дальнего (FAR) вызова. Несоблюдение этого правила и смешение двух разных моделей памяти приведет к повреждению стека и аварийному завершению вашей программы. Кроме того, вы должны использовать дальние (FAR) указатели при ссылке на глобальные данные.

Ручная оптимизация

Одна из основных причин, по которым используется код на ассемблере, заключается в том, что он позволяет осуществлять ручную оптимизацию кода. Ни один, даже самый совершенный компилятор, не может постоянно генерировать код, который был бы лучше созданного опытным программистом на ассембле

ре. Рекомендуется следующий метод повышения производительности функции: используйте код на ассемблере, сгенерированный компилятором, в качестве начальной точки, после чего вручную оптимизируйте функцию. Используя этот подход, вы не должны будете фактически писать программу на ассемблере от начала и до конца. Вместо этого большую часть работы за вас выполнит компилятор, а вы только подправите и усовершенствуете созданный им код.

На первый взгляд вы могли бы подумать, что ручная оптимизация применима только к большим или сложным функциям, но это не так. От ручной оптимизации выигрывают многие и многие функции, так как даже оптимизирующие компиляторы не всегда эффективно используют регистры. Рассмотрим, например, следующий фрагмент кода одного из вышеприведенных примеров, в котором в качестве параметра вызываемой функции использовалась ссылка:

```
_main   proc near
        enter    4,0
     ;
     ;  {
     ;  int num;
     ;
     ;
     ;  get_val(num);
     ;
        lea  ax,word ptr [bp-2]
        push ax
        call near ptr @get_val$qpi
        pop  cx
     ;
     ;  cout << num;
     ;
        mov  ax, word ptr [bp-2]
        mov  word ptr [bp-4],ax
        mov  ax, word ptr [bp-4]
        cwd
        push dx
        push ax
        push offset DGROUP:_cout
        call near ptr @ostream@$blsh$ql
        add  sp,6
     ;
     ;
     ;  return 0;
     ;
        xor ax,ax
        leave
        ret
```

```
    ;
    ;    }
    ;
         leave
         ret

_main    endp
```

В этом примере избыточными являются следующие строки кода:

```
mov  ax, word ptr [bp-2]
mov  word ptr [bp-4],ax
mov  ax, word ptr [bp-4]
```

Совершенно очевидно, что нет необходимости перемещать AX в стек по адресу [bp-4], чтобы сразу же переместить эти данные обратно в AX (следующая строка кода). Поэтому третью строку этого кода можно удалить, благодаря чему код станет компактнее и быстрее. Попробуйте сделать это сами.

Есть еще один метод оптимизации этой функции. Обратите внимание, что когда функция **main** завершает выполнение (*return*) и передает управление операционной системе, в коде дважды повторяется последовательность инструкций LEAVE и RET. Вторая последовательность этих инструкций совершенно не обязательна. Удаление этих строк из кода сделает его более компактным. Следовательно, как вы сами можете убедиться, даже для небольших функций ручная оптимизация может иметь смысл.

Построение основы для кода на ассемблере

Проанализировав вышеприведенные примеры, вы можете взять на вооружение очень полезный метод написания собственных функций на ассемблере: пусть основу ассемблерного кода вашей функции создает компилятор. Получив эту основу, вы можете откорректировать детали. Предположим, например, что вам требуется создать процедуру на ассемблере, которая выполняет умножение двух целых. Для того, чтобы компилятор построил вам основу этой функции, сначала создайте файл, содержащий пустую функцию:

```
mul(int a, int b)
{
}
```

Затем откомпилируйте этот файл с использованием соответствующих опций компиляции. Вы получите файл на языке ассемблера. Например, если вы используете 16-разрядный компилятор Borland, вы получите следующий результат:

```
          .286p
          ifndef    ??version
?debug    macro
          endm
publicdll macro     name
          public    name
          endm
$comm     macro     name,dist,size,count
          comm dist name:BYTE:cout*size
          endm
          else
$comm     macro     name,dist,size,count
          comm dist name[size]:BYTE:count
          endm
          endif
          ?debug    V  301h
          ?debug    S  "mul.cpp"
_TEXT     segment   byte public 'CODE'
_TEXT     ends
DGROUP    group     _DATA,_BSS
          assume    cs:_TEXT,ds:DGROUP
_DATA     segment   word public 'DATA'
d@        label     byte
d@w       label     word
_DATA     ends
_BSS      segment   word public 'BSS'
b@        label     byte
b@w       label     word
_BSS      ends
_TEXT     segment   byte public 'CODE'
    ;
    ;     mul(int a, int b)
    ;
          assume    cs:_TEXT,ds:DGROUP
@mul$qii  proc near
          push bp
          mov  bp,sp
    ;
    ;     {
    ;     }
    ;
```

```
         pop  bp
         ret
@mul$qii endp
         ?debug   C E9
         ?debug   C FA00000000
_TEXT    ends
_DATA    segment word public 'DATA'
@s       label byte
_DATA    ends
_TEXT    segment byte public 'CODE'
_TEXT    ends
_s@      equ  s@
         public   @mul$qii
         end
```

Получив эту основу, вы можете детально разрабатывать функцию. Завершенный пример функции умножения целых может выглядеть, например, так:

```
           .286p
           ifndef    ??version
?debug     macro
           endm
publicdll  macro     name
           public    name
           endm
$comm      macro     name,dist,size,count
           comm dist name:BYTE:cout*size
           endm
           else
$comm      macro     name,dist,size,count
           comm dist name[size]:BYTE:count
           endm
           endif
           ?debug    V  301h
           ?debug    S  "mul.cpp"
_TEXT      segment   byte public 'CODE'
_TEXT      ends
DGROUP     group     _DATA,_BSS
           assume    cs:_TEXT,ds:DGROUP
_DATA      segment   word public 'DATA'
d@         label     byte
d@w        label     word
_DATA      ends
_BSS       segment   word public 'BSS'
```

```
b@        label    byte
b@w       label    word
_BSS      ends
_TEXT     segment  byte public 'CODE'
     ;
     ;    mul(int a, int b)
     ;
          assume   cs:_TEXT,ds:DGROUP
@mul$qii  proc near
          push bp
          mov  bp,sp
          mov  ax, word ptr [bp+4]
          imul word ptr [bp+6]
          pop  bp
          ret
@mul$qii  endp
          ?debug   C E9
          ?debug   C FA00000000
_TEXT     ends
_DATA     segment word public 'DATA'
@s        label byte
_DATA     ends
_TEXT     segment byte public 'CODE'
_TEXT     ends
_s@       equ  s@
          public   @mul$qii
          end
```

Как можно видеть из этого примера, функция **mul** была превращена в полноценную функцию на ассемблере путем добавления всего лишь двух строк кода. Всю остальную работу проделал за вас компилятор.

Если ваша программа на ассемблере использует локальные переменные, вам потребуется выделять для них место в стеке. Для этого вычтите нужное количество байт из SP после его сохранения в BP. Затем, для доступа к локальной переменной, соответствующим образом проиндексируйте стек, используя отрицательное смещение по отношению к BP. Не забывайте сохранять регистры, которые использует ваша функция.

Наилучшим способом узнать больше об интерфейсе кода на ассемблере с вашими программами на C++ является следующий метод: напишите короткие функции на C++, которые должны делать примерно то же, что вы ожидаете от своих функций на ассемблере. Затем откомпилируйте эти файлы с указанием соответствующих опций компилятора и проанализируйте полученный файл на ассемблере. В большинстве случаев вам потребуется ручная оптимизация этого кода, а не написание нового модуля "от нуля".

Использование asm

В ряде случаев существует более простой метод связать код на ассемблере с программой на С++. Возможно, вы уже знаете, что язык С++ обладает inline-возможностями, которые позволяют коду на ассемблере быть частью программы на С++ (для этого используется ключевое слово **asm**). Эта возможность предоставляет двоякие преимущества. Во-первых, вам не требуется писать весь код интерфейса для каждой из таких функций. Во-вторых, весь код будет находиться в одном месте, что несколько упростит его поддержку.

Для того, чтобы вставить в программу код на ассемблере, необходимо предварить ассемблерные инструкции ключевым словом **asm**. Таким образом, каждая строка, которая содержит код на ассемблере, должна начинаться с этого ключевого слова. Компилятор С++ просто передаст эти строки, оставив их без изменений, на следующую, ассемблирующую фазу компиляции.

К примеру, нижеприведенная короткая функция **init_port1()** перемещает значение 88 в регистр AX и выводит его в порты 20 и 21:

```
void init_port1()
{
     cout << "Initializing Port\n";
asm  mov AX, 88
asm  out 20, AX
asm  out 21, AX
}
```

Компилятор автоматически обеспечит интерфейс с кодом, сохранит значения регистров и возвратит управление вызвавшей функции. С вашей стороны необходимо обеспечить только код, выполняющийся внутри функции.

Таким способом вы можете создать inline-версию функции **mul( )**, рассматривавшейся выше, без написания файла на ассемблере. Код этой функции, реализованной с помощью данного подхода, приведен ниже:

```
mul(int a, int b)
{
asm  mov ax, word ptr [bp+4]
asm  imul word ptr [bp+6]
}
```

Не забывайте, компилятор С++ обеспечивает всю необходимую поддержку для вызова функций и возврата управления. Вам необходимо только построить тело функции в соответствии с соглашениями о доступе к аргументам.

Имейте в виду, что какой бы метод вы ни использовали, вы создаете зависимость от конкретной аппаратуры, что затруднит впоследствии портирование вашего кода на другие платформы. Однако, в тех ситуациях, когда создание кода на ассемблере действительно необходимо, ваши усилия будут оправданны.

Рекомендации для самостоятельной разработки

Первое, что следует попробовать реализовать, — это генерация кода на ассемблере с помощью вашего компилятора. Сравните то, что получится у вас с примерами, приведенными в этой главе. Генерирует ли ваш компилятор лучший код по сравнению с приведенным здесь? Имейте в виду, даже различные версии одного и того же компилятора создадут различный код на ассемблере. В качестве следующего шага попробуйте создать код на ассемблере для каждой из моделей памяти, поддерживаемых вашим компилятором. Изучите различия в полученных версиях кода.

Если вы уверены в своей способности программировать на ассемблере, попробуйте вручную оптимизировать несколько критичных функций из ваших собственных приложений. Вполне возможно, вы будете сильно поражены полученным результатом.

Глава 9

Создание и интеграция новых типов данных

Используя С++, вы можете определить новые типы данных и полностью интегрировать их в свою среду программирования. Эта возможность носит названия расширяемости типов (*type extensibility*) и является одной из важнейших, но часто недооцениваемых возможностей С++. После того, как вы полностью определите новый тип данных, он будет выглядеть и действовать в точности так же, как один из встроенных типов. Эта способность к расширению среды С++ путем добавления новых типов данных добавляет к программированию новое "измерение", отсутствующее в других распространенных языках программирования.

Хотя расширяемость типов связана с объектно-ориентированным подходом, она не представляет собой то, о чем обычно думают, как об "объектно-ориентированном программировании". Например, термин "объектно-ориентированное приложение" чаще всего напоминает о масштабных иерархиях классов, виртуальных функциях и абстрактных классах. Расширяемость типов связана с более простым применением объектно-ориентированного программирования. При добавлении нового типа данных вы, как правило, имеете дело с одним классом, для которого требуется определить различные конструкторы, операторы и преобразования. Цель всего этого заключается в построении нового низкоуровневого типа данных, который после этого можно будет использовать так же, как и любой встроенный тип данных. Расширяемость типов не ставит целью создание более крупного приложения. Наиболее иллюстративным примером использования свойства расширяемости типов может служить уже рассмотренный ранее (см. главу 6) класс **string**.

В этой главе мы рассмотрим методы создания новых типов данных и их полной интеграции в состав С++. Нашу разработку мы проведем на примере типа данных **set**. Как, вероятно, вы уже знаете, некоторые языки программи

рования (например, Modula-2) включают в свой состав встроенный тип множества. Что касается C++, то там такого типа данных нет. (Стандартная библиотека шаблонов C++, которая на текущий момент находится в стадии разработки, включает тип данных **set**, но он отличается по своему поведению от аналогичного типа, определенного в этой главе.) Тип **set** мы создадим для хранения множеств элементов и выполнения различных операций над ними. Например, этот тип реализует классические операции над множествами, такие, как объединение, пересечение и симметричную разность.

Хотя в определении нового типа нет ничего фундаментально сложного, для его осуществления все же требуется некоторое количество стандартных шагов. Например, следует определить все операторы, применимые к новому типу данных, включая такие операции, как ввод, вывод и присваивание. Фактически количество кода, необходимое для добавления к C++ даже такого простого типа данных, как множества, не может не вызывать удивления. Однако, проделав всю необходимую работу, вы получите расширение C++ с помощью вашего собственного типа данных.

Теория множеств

Прежде, чем мы начнем реализацию типа данных **set**, необходимо в точности понимать, что же должен представлять собой этот новый тип данных. В соответствии с целями данной главы будем считать множеством набор уникальных элементов. Таким образом, никакие два элемента в составе множества не могут иметь одинаковое значение. Например, в соответствии с принятым определением, нижеприведенные данные будут представлять собой допустимое множество:

{A, B, C}

Все элементы этого множества уникальны. Нижеприведенный набор элементов не является допустимым множеством, так как в нем встречаются дублирующиеся элементы:

{A, B, C, A}

Примечание: Традиционным методом обозначения множеств является заключение их элементов в фигурные скобки. Эти фигурные скобки, разумеется, не имеют ничего общего с фигурными скобками, которые используются в программах на C++.

Порядок следования элементов множества значения не имеет. К примеру, следующие два множества являются эквивалентными:

{A, B, C}

{C, B, A}

К множествам применим стандартный принцип исключения. Это означает, что конкретный элемент или является членом множества, или нет. Одновременное выполнение обоих этих условий невозможно.

Множество может быть пустым. Пустое множество также называется нулевым (*null set*).

Множество является *подмножеством* другого множества, если в этом другом множестве можно найти все элементы, имеющиеся в первом множестве. Например, множество

{A, B}

является подмножеством множества

{A, B, C}

Соответственно, множество считается *надмножеством* другого множества, если оно содержит все элементы этого второго множества.

Каждый отдельный элемент является членом множества, если он содержится в числе элементов множества. Например, A является членом множества:

{A, B, C}

Операции над множествами

Ко множествам применим ряд операций. Тип множества, определенный в данной главе, реализует следующие операции:

- Объединение
- Пересечение
- Разность
- Симметричную разность

Ниже приведены объяснения каждой из этих операций. Будем предполагать, что существуют следующие множества:

Множество S1: {A, B, C}

Множество S2: {C, D, E}

Результатом *объединения* двух множеств будет третье множество, в котором содержатся элементы обоих множеств-операндов S1 и S2:

{A, B, C, D, E}

Обратите внимание на то, что элемент C входит в состав нового множества только один раз, так как множество не допускает дублирования элементов

Пересечением двух множеств является третье, в состав которого войдут только те элементы, которые входят как в состав первого множества, так и в состав второго. Например, результатом пересечения множеств S1 и S2 будет множество, состоящее только из одного элемента:

{C}

Так как C является единственным элементом, входящим в состав как S1, так и в состав S2, он и будет единственным членом множества, являющегося пересечением S1 и S2.

Разницей (иногда называемой разностью) двух множеств будет новое множество, содержащее элементы первого множества, не входящие в состав второго. Например, в результате выполнения операции S1—S2 будет получено следующее множество:

{A, B}

Поскольку C входит также в состав S2, этот элемент вычитается из состава S1 и не входит в состав результирующего множества.

Симметричной разностью между двумя множествами будет новое множество, состоящее из элементов, которые входят в состав одного или другого множества (но не обоих). Например, симметричная разность множеств S1 и S2 представляет собой следующее множество:

{A, B, D, E}

Как вы можете видеть, C не входит в состав результирующего множества, так как является членом как S1, так и S2.

Определение типа множества

Разработанный в данной главе тип set реализует множества в соответствии с вышеописанными правилами. Он использует приведенный ниже класс **Set**:

```
template <class Stype> class Set {
     Stype *SetPtr; // указатель на члены множества
     int MaxSize; // максимальный размер множества
     int NumMembers; // количество элементов множества
     void insert(Stype member); // добавление элемента
     void remove(Stype member); // удаление элемента
     int find(Stype member); // возвращение индекса элемента
     int ismemeber(Stype member); // TRUE, если member член множества
public:
     Set();
     Set(int size);
```

```
    Set(const Set &ob);
    ~Set() {delete SetPtr;}

    Set<Stype> &operator=(Set<Stype> &ob); // присваивание

    // Добавление нового элемента
    Set<Stype> operator+(Stype member);
    friend Set<Stype> operator+(Stype member, Set<Stype> ob);

    Set<Stype> operator+(Set<Stype> &ob); // создание объединения

    Set<Stype> operator-(Stype member); // удаление члена
    Set<Stype> operator-(Set<Stype> &ob); // разница

    Set<Stype> operator&(Set<Stype> &ob); // пересечение
    Set<Stype> operator^(Set<Stype> &ob); // симметричная разность

    // реляционные операции
    int operator==(Set<Stype> &ob); // TRUE в случае равенства
    int operator!+(Set<Stype> &ob); // TRUE в случае неравенства
    int operator<(Set<Stype> &ob); // TRUE в случае подмножества

    // TRUE если член является частью ob
    friend int operator<(Stype member, Set<Stype> ob);

    // преобразование в int возвращает количество членов множества
    operator int() {return NumMembers;}

    // ввод и вывод членов множества
    friend istream &operator<<(istream &stream, Set<Stype> &ob);
    friend ostream &operator<<(ostream &stream, Set<Stype> &ob);
};
```

Как видно из вышеприведенного листинга, **Set** представляет собой параметризованный класс. Это означает, что его можно использовать для построения множеств данных любого типа.

Set содержит три члена, представляющих собой данные: **SetPtr**, **MaxSize** и **NumMembers**. Память для хранения элементов множества выделяется динамически, а указатель на эту память хранится в переменной **SetPtr**. Максимальное количество членов, которое может содержаться в составе множества, хранится в **MaxSize**. Количество элементов, которое содержит множество на текущий момент, хранится в переменной **NumMembers**. Эти члены класса **Set** являются защищенными и не могут быть изменены кодом пользовательского уровня.

Последующие несколько разделов подробно разбирают работу класса **Set**.

Конструкторы и деструктор класса Set

Множества можно декларировать одним из двух способов. Во-первых, они могут создаваться с использованием размера множества по умолчанию. Во-вторых, их можно декларировать как имеющие определенный размер. В этом случае размер множества должен передаваться конструктору в качестве параметра. Оба конструктора класса **Set** приведены ниже:

```
// Построение множества с использованием размера по умолчанию
template <class Stype> Set<Stype>::Set()
{
     SetPtr = new Stype[DEFSET]
     if(!SetPtr) {
          cout << "Allocation error.\n";
          exit(1);
     }
     NumMembers = 0;
     MaxSize = DEFSET;
}

// Построение множества заданного размера
template <class Stype> Set<Stype>::Set(int size)
{
     SetPtr = new Stype[size]
     if(!SetPtr) {
          cout << "Allocation error.\n";
          exit(1);
     }
     NumMembers = 0;
     MaxSize = size;
}
```

Значение **DEFSET** имеет тип **const int** и определяется в вашей программе. Для примера, рассматриваемого в этой главе, используется значение 100. Важно понимать, что значение, содержащееся в переменной **MaxSize**, определяет только максимальное количество элементов, которое может содержаться в множестве. При этом множество фактически может содержать и меньшее количество элементов.

Деструктор **Set** определяется как inline-функция в пределах класса **Set**. Эта функция просто освобождает память, которая была выделена при создании множества.

Добавление и удаление членов множества

Все создаваемые множества изначально пусты. Для того, чтобы добавить в состав множества новый элемент, класс **Set** использует приведенную ниже защищенную функцию-член **insert( )**:

```
// Вставка нового элемента
template <class Stype> void Set<Stype>::insert(Stype member)
{
     if(NumMembers == MaxSize) {
          cout << "Set is full.\n";
          exit(1);
     }

     if(!ismember(member)) {
          // такого элемента в составе множества еще нет
          SetPtr[NumMembers] = member; // добавить
          NumMembers++;
     }
}
```

При вызове функции **insert( )** ей передается в качестве параметра новый член множества. Если множество уже заполнено, будет выведено сообщение об ошибке. В противном случае, если новый элемент еще не является частью множества, он будет добавлен в его состав. Следует помнить о том, что множества не могут содержать дублирующихся элементов. Для того, чтобы определить, не находится ли элемент уже в составе множества, используется функция **ismember( )**. (Описание этой функции см. далее).

Для удаления элемента из множества класс **Set** использует защищенную функцию-член **remove( ):**

```
// Удаление элемента
template <class Stype> void Set<Stype>::remove(Stype member)
{
     int loc = find(member);

     if(loc != -1) {
          // элемент найден
          for(; loc<NumMembers-1; loc++)
               SetPtr[loc] = SetPtr[loc+1];
          NumMembers-;
     }
}
```

Эта функция использует функцию **find( )** для получения индекса элемента, который должен быть удален. Функция **find( )** обсуждается в следующем разделе. Если элемент существует, он "удаляется" из множества путем сдвига всех последующих элементов множества на одну позицию влево, благодаря чему выполняется запись поверх удаляемого элемента. Обратите внимание на то, что переменная **NumMembers** также соответствующим образом уменьшается.

Как уже упоминалось, функции **insert( )** и **remove( )** являются защищенными функциями-членами класса **Set**. Они реализуют низший уровень операций над множествами. Поэтому они предназначены для использования классом **Set**, но не его пользователями. Пользователи **Set** для добавления и удаления элементов должны использовать операторы.

Определение членства

Класс **Set** определяет принадлежность элемента к множеству, вызывая функции **find( )** или **ismember( )**. Эти защищенные функции-члены приведены ниже:

```
/* Найти элемент и возвратить его индекс, если он есть в
составе множества. В противном случае возвратить -1.
*/
template <class Stype> int Set<Stype>::find(Stype member)
{
     int i;

     for(i=0; i<NumMembers; I++)
          if(SetPtr[i] == member) return I;

     return -1;
}

// Вернуть TRUE, если элемент принадлежит множеству
template <class Stype> int Set<Stype>::ismember(Stype member)

{
     if(find(member) != -1) return 1;
     else return 0;
}
```

Функция **find( )** возвращает индекс указанного элемента, если этот элемент находится в составе множества. В противном случае функция возвратит -1. Функция **find( )** существует главным образом для обслуживания функции **remove()**, обсуждавшейся в предыдущем разделе. В большинстве случаев знать фактиче

ский индекс элемента в составе множества не обязательно, и именно на этот случай и предусмотрена функция **ismember( )**. Она возвращает TRUE, если элемент присутствует в составе множества, и FALSE — в противном случае.

Как и в случае с функциями **insert( )** и **remove( )**, функции **find( )** и **ismember( )** предназначены для внутреннего использования класса **Set**. Пользователи класса **Set** для определения членства элементов в составе множества должны использовать операторы.

Конструктор Copy

Класс **Set** требует определения конструктора копирования. Как, возможно, вы уже знаете, конструктор копирования класса вызывается всякий раз, когда выполняется копирование объектов, принадлежащих к этому классу. В частности, конструктор копирования вызывается, когда объект передается функции по значению, при построении временного объекта как возвращаемого значения функции, а также при использовании объекта для инициализации другого объекта. Если класс не содержит явным образом определенного конструктора копирования, то в этом случае при возникновении одной из этих трех ситуаций по умолчанию используется побитовое копирование объекта. Следует отметить, что побитовое копирование) не во всех случаях является адекватным. Именно для таких случаев и необходимо определять собственный конструктор копирования).

Причину, по которой классу **Set** необходим собственный конструктор копирования, понять несложно. Вспомните, что память для каждого множества выделяется с помощью **new**, а указатель на эту память хранится в переменной **SetPtr**. При побитовом копировании указатель, содержащийся в переменной **SetPtr** объекта-копии, будет указывать на ту же самую память, что и указатель **SetPtr** оригинала. Таким образом, оба объекта для хранения множеств будут использовать одну и ту же область памяти. Это, в свою очередь, означает, что изменения в одном из этих объектов повлекут за собой изменения в другом. Если один из объектов будет удален, то память, соответственно, будет освобождена, а ведь она используется другим объектом, который не должен быть удален! Такие ситуации, как правило, приводят к краху программы. К счастью, этого легко можно избежать, если определить конструктор копирования. Конструктор copy класса **Set** должен гарантировать, что указатели **SetPtr** копии и оригинала указывают каждый на свою область памяти. Для этого конструктор копирования выделяет память для хранения нового множества при копировании объекта. Листинг конструктора копирования приведен ниже:

```
// Конструктор копирования
template <class Stype> Set<Stype>::Set(const Set<Stype> &ob)
{
```

```
    int i;

    MaxSize = ob.MaxSize;

    SetPtr = new Stype[MaxSize];
    if(!SetPtr) {
        cout << "Ошибка выделения памяти\n";
        exit(1);
    }

    NumMembers = 0;

    for(i=0; i<ob.NumMembers; i++)
        insert(ob.SetPtr[i]);
}
```

Как видите, конструктор копирования выделяет память для копии, а затем копирует в новый объект каждый элемент исходного. Таким образом, указатели **SetPtr** исходного объекта и копии указывают на разные области памяти.

Присваивание для множеств

Как вы уже знаете, если класс не перегружает оператора присваивания, то при присваивании одному объекту значения другого по умолчанию будет выполняться побитовое копирование. Однако, как для рассматриваемого здесь класса **Set**, так и для большинства реальных классов, этот метод является неприемлемым. Поэтому для класса **Set** необходимо создать собственный оператор присваивания (путем его перегрузки). Причина этого аналогична причине, по которой необходимо определять конструктор копирования. При побитовом копировании вы получите два объекта, использующие одну и ту же область памяти для хранения своих данных. Во избежание такой ситуации оператор присваивания должен просто скопировать содержимое одного множества в другое, не изменяя при этом значения переменной **SetPtr** каждого из этих множеств. Эта задача выполняется приведенной ниже функцией **operator=( )**.

```
// Перегрузка оператора присваивания для множеств
template <class Stype> Set<Stype>
&Set<Stype>::operator=(Set<Stype> &ob)
{
    int i;

    // обработка случая s = s
    if(SetPtr == ob.SetPtr) return *this;
```

```
     // проверяем размер
     if(ob.NumMembers > MaxSize) {
          delete SetPtr;
          SetPtr = new Stype[ob.Numembers];
               if(!SetPtr) {
                    cout << "Ошибка выделения памяти\n";
                    exit(1);
               }
               MaxSize = ob.NumMembers;
          }
          NumMembers = 0; // удаляем старое множество

     for(i=0; i<ob.NumMembers; i++)
          insert(ob.SetPtr[i]);

     return *this;
}
```

Рассмотрим эту функцию более подробно. Во-первых, обратим внимание на тот факт, что если члены **SetPtr** обоих объектов указывают на одну и ту же память, то никаких действий не предпринимается. Причина этого заключается в необходимости обработки частного случая, когда объекту присваивается его собственное значение. Хотя впрямую такую ситуацию никто не создаст, она все же может возникнуть косвенно. В любом случае метод реализации перегрузки оператора присваивания **operator=( )** предусматривает эту ситуацию как частный случай.

Далее, если объект, которому присваивается новое значение, недостаточно велик для того, чтобы в нем могло уместиться присваиваемое значение, то ассоциированная с ним память освобождается, после чего необходимый объем памяти выделяется заново. После этого переменная **NumMembers** инициализируется нулем. Эта операция удаляет все элементы, которые ранее содержались в объекте, которому присваивается новое значение, после чего вставляет в него все элементы исходного множества. Наконец, функция возвращает значение ***this**. Это необходимо для того, чтобы оператор присваивания можно было использовать в более сложных выражениях.

Перегрузка оператора +

Класс **Set** перегружает + для двух типов операций. Первая операция использует + для добавления элемента в состав множества. Вторая операция создает объединение множеств. Обе эти операции описаны ниже.

Добавление нового элемента в состав множества

Использование + для добавления элемента в состав множества требует двух незначительно различающихся версий функции **operator+()**. Первая из них обрабатывает ситуацию множество + *элемент,* а вторая — противоположную ситуацию *(элемент* + *множество*). Обе версии приведены ниже.

```
// set = set + item
template <class Stype> Set<Stype> Set<Stype>::operator+(Stype
member)
{
     int i;
     Set<Stype> temp(NumMembers + 1);

     // Копирование существующих элементов во временное мно-
жество
     for(i=0; i<NumMembers; i++)
          temp.insert(SetPtr[i]);

     // Вставляем новый элемент
          temp.insert(member);

     // Возвращаем новое значение
     return temp;
}

// set = item + set

template<class Stype> Set<Stype> operator+(Stype member,
Set<Stype> ob)
{
     int i;
     Set<Stype> temp(ob.NumMembers + 1);

     // Копирование существующих элементов во временное мно-
жество
     for(i=0; i<ob.NumMembers; i++)
          temp.insert(ob.SetPtr[i]);

     // Вставляем новый элемент
          temp.insert(member);

     // Возвращаем новое значение
     return temp;
```

Работа этих функций проста. Первая из них создает временное множество под именем **temp**, которое достаточно велико для того, чтобы содержать исходное множество плюс один дополнительный элемент. Далее исходное множество копируется в **temp**. Затем в это новое множество **temp** вставляется новый элемент. Наконец, функция возвращает значение **temp**.

Первая версия функции **operator+( )** позволяет использовать для добавления элементов следующее выражение:

set = set + item;

Здесь левый операнд представляет собой множество, а правый — добавляемый элемент.

Вторая версия функции **operator+( ),** которая представляет собой скорее дружественную (**friend**) функцию, а не функцию-член, обрабатывает случай, кода левый операнд представляет собой добавляемый элемент, а правый — множество, к которому он добавляется. Таким образом, эта функция обрабатывает выражения следующего типа:

set = item + set;

Поскольку добавляемый элемент может принадлежать к одному из встроенных типов, необходимо использовать дружественную (**friend**) функцию для того, чтобы эта вторая форма была допустимой. При использовании функции-члена левый операнд передается неявно через параметр **this**. Это означает, что левый операнд функции-члена должен принадлежать к классу, для которого определена эта функция. Таким образом, для того, чтобы элемент (не являющийся объектом класса **Set**) можно было использовать в качестве первого операнда, необходимо использовать дружественную функцию, а не функцию-член. Использование дружественной функции позволяет определять левый операнд как принадлежащий к типу, отличному от **Set**.

Построение объединения множеств

Вторая операция, для которой перегружается +, создает объединение двух множеств. Иными словами, эта операция “складывает” два множества. Листинг этой функции приведен ниже:

```
// Перегрузка сложения для множеств. Построение объединения
template <class Stype> Set<Stype>
Set<Stype>::operator+(Set<Stype> &ob)
{
    int i;
    Set<Stype> temp(NumMembers +ob.NumMembers);

    for(i=0; i<NumMembers; i++)
        temp.insert(SetPtr[i]);
```

```
    for(i=0; i<ob.NumMembers; i++)
        temp.insert(ob.SetPtr[i]);

    return temp;
}
```

Эта функция используется для выполнения операций следующего типа:

set1 = set2 + set3;

После выполнения этого утверждения **set1** будет содержать объединение множеств **set2** и **set3**. Сначала функция создает временное множество под названием **temp**. Это множество должно иметь такой размер, чтобы в нем могли содержаться все элементы, принадлежащие к двум множествам, над которыми выполняется операция. Хотя результат выполнения операции объединения будет содержать такое количество элементов только в том случае, если объединяемые множества не содержат общих элементов, функция должна предусматривать и этот случай. Затем функция добавляет в **temp** содержимое обоих операндов. Здесь уместно напомнить, что функция **insert( )** не добавит в результирующее множество дублирующиеся элементы. Именно по этой причине для функции **operator+( )** нет необходимости выполнять проверку дублирования элементов.

Перегрузка оператора –

Оператор – также перегружается для двух незначительно различающихся операций. Первая из них удаляет (иными словами, вычитает) элемент из множества. Вторая создает разность двух множеств. Обе функции подробно рассматриваются ниже.

Удаление элемента из множества

Для удаления элемента из множества используется функция **operator-( )**.

```
/*
    Перегрузка вычитания. Эта функция удаляет элемент из
множества
*/
template <class Stype> Set<Stype> Set<Stype>::operator-(Stype
member)
{
    int i;
    Set<Stype> temp = *this;
```

```
    temp.remove(member);

    return temp;
}
```

Эта функция позволяет выполнять выражения следующего типа:

set1 = set2 - item;

После выполнения операции результирующее множество **set1** будет содержать все элементы исходного множества **set2**, за исключением элемента **item**.

Разность множеств

Следующая форма функции **operator–( )** вычисляет разность множеств, выполняя вычитание содержимого одного множества из содержимого другого. Эта версия функции **operator–( )** приведена ниже:

```
/*
    Перегрузка вычитания. Эта функция осуществляет вычитание
множеств
*/
template <class Stype> Set<Stype> Set<Stype>::operator-
(Set<Stype> &ob)
{
    int i;
    Set<Stype> temp = *this;

    // удаляем члены, общие для *this и ob

    for(i=0; i<NumMembers; i++) {
        if(ob.ismember(SetPtr[i]))
            temp.remove(SetPtr[i]);
    }
    return temp;
}
```

Эта версия осуществляет такие операции, как:

set1 = set2 – set3;

После выполнения этой операции, множество **set1** будет содержать элементы **set2**, не являющиеся частью **set3**. Функция начинает работу, инициализируя временное множество **temp** переменной ***this** (значение этой переменной представляет собой множество, находящееся слева от знака минус "–". После этого функция удаляет из этого множества все члены, которые принадлежат также и ко второму множеству.

Пересечение множеств

Пересечение двух множеств реализуется оператором **&**. Эта функция приведена ниже:

```
//Пересечение множеств
template <class Stype> Set<Stype>
Set<Stype>::operator&(Set<Stype> &ob)
{
     int i, j;
     Set<Stype> temp(NumMembers);

     for(i=o; i<NumMembers; i++) {
          if(ob.ismember(SetPtr[i]))
               temp.insert(SetPtr[i]);
     }
     return temp;
}
```

Эта функция сначала создает временный объект, имеющий размер, достаточный для того, чтобы в нем могло поместиться самое большое пересечение. После этого функция копирует в этот временный объект элементы, общие для обоих множеств.

Оператор **&** может использоваться в следующих типах выражений:

set1 = set2 & set3;

После выполнения этого утверждения, множество **set1** будет содержать пересечение множеств **set2** и **set3**.

Симметричная разность

Симметричная разность двух множеств представляет собой разность их объединения и пересечения. Она реализуется с помощью оператора ^. Листинг этой функции приведен ниже:

```
/*
     Симметричная разность двух множеств
*/
template <class Stype> Set<Stype>
Set<Stype>::operator^(Set<Stype> &ob)
{
     int i, j;
     Set<Stype> temp1, temp2;
```

```
    temp1 = *this + ob;
    temp2 = *this & ob;
    temp1 = temp1 - temp2;

    return temp1;
}
```

Принцип действия этой функции прост и понятен. Сначала функция создает временное множество, состоящее из объединения двух исходных множеств. Затем создается второе временное множество, содержащее пересечение первых двух. Наконец, функция вычитает пересечение из объединения.

Оператор ^ может использоваться в выражениях следующего вида:

set1 = set2 ^ set3;

После выполнения этого утверждения **set1** будет содержать симметричную разность **set2** и **set3**.

Определение равенства, неравенства и подмножества

Класс **Set** содержит операторы, определяющие равенство и неравенство двух множеств, а также оператор, указывающий на то, что одно множество является подмножеством другого. Равенство и неравенство для класса **Set** реализованы перегрузкой операторов **==** и **!=** соответственно. Статус подмножества задается оператором **<**. Соответствующие функции-операторы перечислены ниже.

```
// Возвратить TRUE, если множества равны
template<class Stype> int Set<Stype>::operator==(Set<Stype>
&ob)
{
    // множества должны содержать одинаковое количество эле
ментов
    if(NumMembers != ob.NumMembers) return 0;

    return *this < ob;
}

// Возвратить TRUE, если множества не равны
template<class Stype> int Set<Stype>::operator!=(Set<Stype>
&ob)
{
    return !(*this == ob);
}
```

```
// Возвратить TRUE, если *this - подмножество ob
template<class Stype> int Set<Stype>::operator<(Set<Stype>
&ob)
{
     int i;

     for(i=0; i<NumMembers; i++)
          if(!ob.ismember(SetPtr[i])) return 0;

     return 1;
}
```

Принцип работы этих функций прост и понятен. Следует, однако, обратить внимание на то, что в функции **operator==( )** сначала выполняется проверка того факта, что множества содержат одинаковое количество элементов. Если это так, то функция, используя оператор подмножества, проверяет, не является ли одно из множеств подмножеством другого. Если два множества имеют одинаковый размер и одно из них является подмножеством другого, то эти множества равны (эквивалентны).

Эти функции позволяют выполнять утверждения, подобные нижеприведенным:

if(set1 == set2) cout << “Множества равны”;
if(set1 != set2) cout << “Множества отличаются”;
if(set1 < set2) cout << “Множество 1 – подмножество множества 2”;

Определение членства

Иногда бывает очень полезно знать, входит ли элемент в состав множества. Для этого класс **Set** перегружает оператор **<** несколько иным способом, описанным ниже:

```
// Возвратить TRUE, если элемент входит в состав множества.
template <class Stype> int operator<(Stype member, Set<Stype>
ob)
{
     return ob.ismember(member);
}
```

Эта функция позволяет выполнять утверждения следующего типа:

if(item<set) cout << “Элемент **item** входит в состав множества **set**”;

Для определения членства элемента в составе множества эта функция использует защищенную функцию член **ismember( )**.

Преобразование в целое

Класс **Set** содержит одну функцию преобразования, обеспечивающую преобразование в целое. В соответствии с данным здесь определением преобразование объекта **Set** в целое возвращает число, равное количеству элементов, содержащихся в составе множества на текущий момент. Если множество пусто, это преобразование возвратит нуль. Как, возможно, вы уже знаете по предшествующему опыту программирования на С++, функция преобразования нужна для автоматического преобразования элементов, имеющих тип класса, к другому (обычно встроенному) типу. Функция преобразования позволяет использовать объекты данного класса в выражениях, содержащих данные других типов. В конкретном случае класса **Set** необходимо задать функцию преобразования в **int**. Для других типов классов, которые вы можете разработать самостоятельно, могут потребоваться различные функции преобразования.

Функция преобразования к целому типу очень проста и коротка. Она определяется как inline-функция в пределах класса **Set**. Ее текст приведен ниже для удобства

```
operator int() {return NumMembers; }
```

Эта функция преобразования очень полезна. К примеру, она позволяет выполнять выражения, подобные нижеприведенным:

```
if(set)    cout << "Множество не пустое";
cout << "set1 содержит " << (int)set1 << "элементов.\n";
```

В первом утверждении сначала происходит преобразование **set** к целому типу, после чего условие, управляющее оператором условного перехода, будет выполнено только в том случае, если множество содержит элементы. Во втором утверждении происходит явное преобразование множества **set1** к типу **int**, после чего отображается количество элементов, содержащихся в множестве.

Перегрузка операторов ввода/вывода

Последними операторами, перегружаемыми классом **Set**, являются операторы ввода/вывода, приведенные ниже:

```
// Ввод
template <class Stype> istream &operator>>(istream &stream,
Set<Stype> &ob)
{
    Stype member;

    stream >> member;
    ob = ob + member;
    return stream;
}
```

```
// Вывод
template class<Stype> ostream &operator << (ostream &stream,
Set<Stype> &ob)
[
     int i;

     for(i=0; i<ob.NumMembers; i++)
          stream <<ob.SetPtr[i] << " ";

     stream << endl;

     return stream;
}
```

Имейте в виду, что эти функции очень просты. Если вы будете создавать сложные типы множеств, вам, вероятно, придется разработать собственные версии функции для обработки ввода/вывода этих типов.

Демонстрационная программа работы с множествами

Нижеприведенная программа содержит полный листинг класса **Set** и функцию **main()**, демонстрирующую различные операции над множествами.

```
// Класс множеств
#include <iostream.h>
#include <stdlib.h>

const int DEFSET = 100;
template <class Stype> class Set {
     Stype *SetPtr; // указатель на члены множества
     int MaxSize; // максимальный размер множества
     int NumMembers; // количество элементов множества
     void insert(Stype member); // добавление элемента
     void remove(Stype member); // удаление элемента
     int find(Stype member); // возвращение индекса элемента
     int ismemeber(Stype member); // TRUE, если member член множества
public:
     Set();
     Set(int size);
     Set(const Set &ob);
     ~Set() {delete SetPtr;}
     Set<Stype> &operator (Set<Stype> &ob); // присваивание
```

```
// Добавление нового элемента
Set<Stype> operator+(Stype member);
friend Set<Stype> operator+(Stype member, Set<Stype> ob);

Set<Stype> operator+(Set<Stype> &ob); // создание объединения

Set<Stype> operator-(Stype member); // удаление члена
Set<Stype> operator-(Set<Stype> &ob); // разница

Set<Stype> operator&(Set<Stype> &ob); // пересечение
Set<Stype> operator^(Set<Stype> &ob); // симметричная разность

// реляционные операции
int operator==(Set<Stype> &ob); // TRUE в случае равенства
int operator!+(Set<Stype> &ob); // TRUE в случае неравенства
int operator<(Set<Stype> &ob); // TRUE в случае подмножества

// TRUE если член является частью ob
friend int operator<(Stype member, Set<Stype> ob);

// преобразование в int возвращает количество членов множества
operator int() {return NumMembers;}

// ввод и вывод членов множества
friend istream &operator<<(istream &stream, Set<Stype> &ob);
friend ostream &operator<<(ostream &stream, Set<Stype> &ob);
};

// Построение множества с использованием размера по умолчанию
template <class Stype> Set<Stype>::Set()
{
     SetPtr = new Stype[DEFSET]
     if(!SetPtr) {
          cout << "Allocation error.\n";
          exit(1);
     }
     NumMembers = 0;
     MaxSize = DEFSET;
}
// Построение множества заданного размера
template <class Stype> Set<Stype>::Set(int size)
{
     SetPtr = new Stype[size]
     if(!SetPtr) {
          cout << "Allocation error.\n";
          exit(1);
     }
```

```
     NumMembers = 0;
     MaxSize = size;
}

// Конструктор копирования
template <class Stype> Set<Stype>::Set(const Set<Stype> &ob)
{
     int i;

     MaxSize = ob.MaxSize;

     SetPtr = new Stype[MaxSize];
     if(!SetPtr) {
          cout << "Ошибка выделения памяти\n";
          exit(1);
     }

     NumMembers = 0;

     for(i=0; i<ob.NumMembers; i++)
          insert(ob.SetPtr[i]);
}

// Вставка нового элемента
template <class Stype> void Set<Stype>::insert(Stype member)
{
     if(NumMembers == MaxSize) {
          cout << "Set is full.\n";
          exit(1);
     }

     if(!ismember(member)) {
          // такого элемента в составе множества еще нет
          SetPtr[NumMembers] = member; // добавить
          NumMembers++;
     }
}
// Удаление элемента
template <class Stype> void Set<Stype>::remove(Stype member)
{
     int loc = find(member);

     if(loc != -1) {
          // элемент найден
          for(; loc<NumMembers-1; loc++)
               SetPtr[loc] = SetPtr[loc+1];
```

```
            NumMembers—;
      }
}

/* Найти элемент и возвратить его индекс, если он есть в
составе множества. В противном случае возвратить -1.
*/
template <class Stype> int Set<Stype>::find(Stype member)
{
      int i;

      for(i=0; i<NumMembers; I++)
            if(SetPtr[i] == member) return I;

      return -1;
}

// Вернуть TRUE, если элемент принадлежит множеству
template <class Stype> int Set<Stype>::ismember(Stype member)

{
      if(find(member) != -1) return 1;
      else return 0;
}

// Перегрузка оператора присваивания для множеств
template <class Stype> Set<Stype>
&Set<Stype>::operator=(Set<Stype> &ob)
{
      int i;

      // обработка случая s = s
      if(SetPtr ==ob.SetPtr) return *this;

      // проверяем размер
      if(ob.NumMembers > MaxSize) {
            delete SetPtr;
            SetPtr = new Stype[ob.NumMembers];
                  if(!SetPtr) {
                        cout << "Ошибка выделения памяти\n";
                        exit(1);
                  }
                  MaxSize = ob.NumMembers;
            }
            NumMembers   0; // удаляем старое множество
```

```
    for(i=0; i<ob.NumMembers; i++)
        insert(ob.SetPtr[i]);

    return *this;
}

// Перегрузка сложения (операция типа set = set + item)
template <class Stype> Set<Stype> Set<Stype>::operator+(Stype
member)
{
    int i;
    Set<Stype> temp(NumMembers + 1);

    // Копирование существующих элементов во временное множество
    for(i=0; i<NumMembers; i++)
        temp.insert(SetPtr[i]);

    // Вставляем новый элемент
        temp.insert(member);

    // Возвращаем новое значение
    return temp;
}

// Перегрузка сложения (операция типа set = item + set)

template<class Stype> Set<Stype> operator+(Stype member,
Set<Stype> ob)
{
    int i;
    Set<Stype> temp(ob.NumMembers + 1);

    // Копирование существующих элементов во временное множество
    for(i=0; i<ob.NumMembers; i++)
        temp.insert(ob.SetPtr[i]);
    // Вставляем новый элемент
        temp.insert(member);

    // Возвращаем новое значение
    return temp;
}

// Перегрузка сложения для множеств. Построение объединения
template <class Stype> Set<Stype>
Set<Stype>::operator+(Set<Stype> &ob)
{
    int i;
    Set<Stype> temp(NumMembers +ob.NumMembers);
```

```
    for(i=0; i<NumMembers; i++)
        temp.insert(SetPtr[i]);
    for(i=0; i<ob.NumMembers; i++)
        temp.insert(ob.SetPtr[i]);

    return temp;
}

/*
Перегрузка вычитания. Эта функция осуществляет вычитание множеств
*/
template <class Stype> Set<Stype> Set<Stype>::operator-
(Set<Stype> &ob)
{
    int i;
    Set<Stype> temp = *this;

    // удаляем члены, общие для *this и ob

    for(i=0; i<NumMembers; i++) {
        if(ob.ismember(SetPtr[i]))
            temp.remove(SetPtr[i]);
    }
    return temp;
}

/*
Перегрузка вычитания. Эта функция удаляет элемент из множества
*/
template <class Stype> Set<Stype> Set<Stype>::operator-(Stype
member)
{
    int i;
    Set<Stype> temp = *this;

    temp.remove(member);

    return temp;
}

//Пересечение множеств
template <class Stype> Set<Stype>
Set<Stype>::operator&(Set<Stype> &ob)
{
    int i, j;
    Set<Stype> temp(NumMembers);
```

```
    for(i=0; i<NumMembers; i++) {
        if(ob.ismember(SetPtr[i]))
            temp.insert(SetPtr[i]);
    }
    return temp;
}

/*
    Симметричная разность двух множеств
*/
template <class Stype> Set<Stype>
Set<Stype>::operator^(Set<Stype> &ob)
{
    int i, j;
    Set<Stype> temp1, temp2;

    temp1 = *this + ob;
    temp2 = *this & ob;
    temp1 = temp1 - temp2;

    return temp1;
}

// Возвратить TRUE, если множества равны
template<class Stype>
int Set<Stype>::operator==(Set<Stype> &ob)
{
    // множества должны содержать одинаковое количество элементов
    if(NumMembers != ob.NumMembers) return 0;
    return *this < ob;
}

// Возвратить TRUE, если множества не равны
template<class Stype>
int Set<Stype>::operator!=(Set<Stype> &ob)
{
    return !(*this == ob);
}

// Возвратить TRUE, если *this - подмножество ob
template<class Stype>
int Set<Stype>::operator<(Set<Stype> &ob)
{
    int i;

    for(i=0; i<NumMembers; i++)
```

```
        if(!ob.ismember(SetPtr[i])) return 0;

    return 1;
}

// Возвратить TRUE, если элемент входит в состав множества.
template <class Stype>
int operator<(Stype member, Set<Stype> ob)
{
    return ob.ismember(member);
}

// Ввод
template <class Stype>
istream &operator>>(istream &stream, Set<Stype> &ob)
{
    Stype member;

    stream >> member;
    ob = ob + member;

    return stream;
}

// Вывод
template class<Stype>
ostream &operator << (ostream &stream, Set<Stype> &ob)
{
    int i;
    for(i=0; i<ob.NumMembers; i++)
        stream <<ob.SetPtr[i] << " ";

    stream << endl;

    return stream;
}

main()
{
    // Множества целых

    Set<int> set1(10), set2(10), set3(10);
    if(set1)
        cout << "set1 contains members.\n";
    else
    cout << "set1 is empty.\n";
```

```
    set1 = set1 + 1; //set + member
    set1 = 2 + set1; //member + set
    set1 = set1 + 3; //set + member
    set1 = 4 + set1; //member + set

    set2 = set2 + 1;
    set2 = set2 + 3;
    set2 = set2 + 5;
    set2 = set2 + 6;

    if(set1)
         cout << "set1 contains members.\n";
    else
         cout << "set1 is empty.\n";

    cout << "Set in set1: ";
    cout << set1;
    cout << "Set in set2: ";
    cout << set2;

    cout << "Union of set1 and set2: ";
    set3 = set1 + set2;
    cout << set3;

    cout << "Intersection of set1 and set2: ";
    set3 = set1 & set2;
    cout << set3;
    cout << "Difference of set1 and set2: ";
    set3 = set1 - set2;
    cout << set3;

    cout << "Symmetric difference of set1 and set2: ";
    set3 = set1 ^ set2;
    cout << set3;
    cout << endl;

    set3 = set1 + set2;
    cout << "set3 now contains set1 + set2: ";
    cout << set3;

    if(1 < set3)
         cout << "1 is a member of set 3.\n";
    if(0 < set3)
         cout << "0 is a member of set3.\n";
    else
         cout << "0 is not a member of set3.\n";
```

```
if(set1 < set3)
     cout << "set1 is a subset of set3.\n";
if(set3 < set1)
     cout << "This will not be printed.\n"
else
     cout << "set3 is not a subset of set1.\n";

set1 = set1 + 99;
cout << "Set1 now contains: ";
cout << set1;
if(set1 < set3)
     cout << "This will not be printed\n";
else
     cout << "Now set1 is not a subset of set3.\n";

cout << "Enter an integer: ";
cin >> set3;
cout << "Set3 now contains: " << set3;

cout << "set3 after removing 1: ";
set3 = set3 -1;
cout << set3;
cout << "set3 after removing 3: ";
set3 = set3 - 3;
cout << set3;
cout << endl;
Set<int> set4 = set1, set5 = set1;
cout << "Here is set4: ";
cout << set4;
cout << "Here is set5: ";
cout << set5;

if(set4 == set5)
     cout << "Sets in set4 and set5 are equal.\n";

set4 = set4 + 30;
cout << "Now, here is set4: ";
cout << set4;
if(set4 != set5)
     cout << "Now set4 and set5 are not equal.\n";

cout << endl;
// set of char * pointers
Set<char *> strset(4);
strset = strset + "one";
strset = strset + "two";
```

```
    cout << "Set of char*: ";
    cout << strset << endl;

    // set of characters
    Set<char> chset1, chset2;

    chset1 = chset1 + 'a';
    chset1 = chset1 + 'b';
    chset1 = chset1 + 'c';

    chset2 = chset1 + 'z';

    cout << "chset1 and chset2:\n";
    cout << chset1 << chset2;
    cout << "Intersection of chset1 and chset2: ";
    cout << (chset1 & chset2);
    cout << "Symmetric difference of chset1 and chset2: ";
    cout << (chset1 ^ chset2);

    return 0;
}
```

Как видно из этого кода, тип **Set** можно использовать точно так же, как и любой другой тип данных. Он полностью интегрирован в среду C++. В частности, выражения, в которые входят данные класса **Set**, выглядят точно так же, как и выражения, включающие встроенные типы. Этот пример иллюстрирует широкие возможности расширения C++. Образец вывода этой программы приведен ниже:

set1 is empty
set contains members
Set in set1: 1 2 3 4
Set in set2: 1 3 5 6
Union of set1 and set2: 1 2 3 4 5 6
Intersection of set1 and set2: 1 3
Difference of set1 - set2: 2 4
Symmetric difference of set1 and set2: 2 4 5 6
set3 now contains set1 + set2: 1 2 3 4 5 6
1 is a member of set3
0 is not a member of set3
set1 is a subset of set3
set3 is not a subset of set1
set1 now contains: 1 2 3 4 99
Now set1 is not a subset of set3
Enter an integer: 2500
set3 now contains: 1 2 3 4 5 6 2500
set3 after removing 1: 2 3 4 5 6 2500
set3 after removing 3: 2 4 5 6 2500

```
Here is set4: 1 2 3 4 99
Here is set5: 1 2 3 4 99
Sets in set4 and set5 are equal
Now here is set4: 1 2 3 4 99 30
Now set4 and set5 are not equal

Set of char*: one two

chset1 and chset2:
a b c
a b c z
Intersection of chset1 and chset2: a b c
Symmetric difference of chset1 and chset2: z
```

Рекомендации для самостоятельной разработки

Здесь приведены некоторые доработки, которые будет полезно внести в класс **Set**. Во-первых, множества должны расти динамически, а не иметь фиксированный размер. Во-вторых, введите оператор-функцию надмножества **operator>( )**. Эта функция должна возвращать TRUE, если множество в левой части выражения является надмножеством множества, расположенной в правой части. Наконец, введите функцию-итератор, которая при каждом последующем вызове будет возвращать следующий элемент множества.

Наконец, попробуйте разрабатывать собственные типы данных и полностью интегрировать их в состав С++. Как видно из материала данной главы, это не потребует больших усилий.

Глава 10

Реализация языковых интерпретаторов на C++

Хотели бы вы разработать собственный язык программирования? Если вы — типичный программист, вам, вероятнее всего, этого хотелось бы. Идея построить, разработать, расширить и модифицировать собственный язык программирования, над которым вы будете обладать полным контролем, привлекает многих программистов. Немногие, однако, понимают при этом, насколько процесс создания собственного языка может быть прост и приятен. Разумеется, разработка полнофункционального компилятора представляет собой серьезный проект, но разработать собственный интерпретатор языка — гораздо более доступная задача, В этой главе мы обсудим секреты интерпретатора языка и создадим его работающий пример.

Интерпретаторы важны по следующим четырем причинам. Во-первых, они способны обеспечить по-настоящему интерактивную среду. Многие приложения, такие, как робототехника, требуют именно интерактивной, а не компилированной среды. Во-вторых, по своей природе языковые компиляторы особенно хорошо подходят для интерактивной отладки. В-третьих интерпретаторы позволяют строить великолепные языки запросов для систем управления базами данных. Фактически многие из таких языков были разработаны именно на базе интерпретаторов. Наконец, интерпретаторы важны еще и потому, что они позволяют одной и той же программе работать на различных платформах. Для переноса программы на новую платформу требуется только реализовать для этой платформы библиотеку модулей времени выполнения (run-time package). Исходный код программы при этом не претерпевает никаких изменений. Например, **Java**), язык апплетов Internet), является интерпретатором именно по этой причине.

Хотя компиляторы всегда будут оставаться на переднем крае программирования, интерпретаторы также приобретают все большее значение. Вполне вероятно, что в ходе продвижения вашей карьеры программиста на C++ вам придется написать несколько интерпретаторов. К счастью, язык C++ идеально подходит для этой цели.

Разумеется, для того, чтобы понять принципы работы интерпретатора, необходимо разработать такой интерпретатор для одного из языков программирования. Вполне очевидным кандидатом был бы сам С++, но, к сожалению, он слишком богат и сложен, и создать для него интерпретатор было бы не такой уж простой задачей. (Исходный код интерпретатора даже для очень ограниченного подмножества языка С++ никогда не уместится в главе книги.) Поэтому технику построения интерпретатора мы рассмотрим на примере подмножества языка BASIC (далее будем называть это подмножество Small BASIC). Мы выбрали BASIC по трем основным причинам. Во-первых, этот язык изначально разрабатывался как интерпретатор. Поэтому реализовать для него интерпретатор будет довольно просто. Например, стандартный BASIC не поддерживает таких понятий, как локальные переменные, рекурсивные функции, блоки, классы, перегрузку, шаблоны и т. д — а все эти понятия вносят свой вклад в возрастание сложности кода интерпретатора. (И именно поэтому реализация интерпретатора для С++ намного сложнее, чем реализация интерпретатора для BASIC.) Тем не менее, все основные принципы, используемые для построения интерпретатора BASIC, применимы и к любому другому языку программирования. Поэтому разработанные в этой главе процедуры могут послужить для вас стартовой точкой при разработке собственных более сложных проектов. Второй причиной выбора языка BASIC явилось то, что интерпретатор для его реального подмножества, представляющего собой полноценный язык программирования, можно реализовать в виде относительно небольшого объема кода. Наконец, BASIC выбран еще и потому, что большинство программистов знают его (хотя бы поверхностно). И даже если вы совершенно не знаете BASIC, не волнуйтесь по этому поводу. Команды, используемые в Small BASIC, тривиально просты.

Примечание: Если вы очень заинтересованы в разработке интерпретаторов, вы найдете описание интерпретатора С в моей книге “С: The Complete Reference”, 3rd Edition (Osborne, McGraw-Hill, 1995). Если вы все же решитесь разрабатывать собственный интерпретатор С++, то приведенный в этой книге интерпретатор С послужит хорошей стартовой точкой для этой разработки.

Модуль разбора выражений Small BASIC

Самой важной частью языкового интерпретатора является модуль разбора выражений. Как вы уже знаете из главы 3, модуль разбора выражений используется для преобразования численных выражений типа (10-X)/23 в форму, доступную компьютеру для интерпретации и вычислений. Поскольку разбор выражений подробно обсуждался в главе 3, повторять его здесь нет необходимости. Тем не менее, модуль разбора выражений, используемый здесь, не является точной копией программы, обсуждавшейся в главе 3. Фундаментальные принципы работы модуля разбора выражений остались неизменными, но функциональные возможности Small BASIC, которые должны быть реализованы интерпретатором, требуют построения его специализированной версии. Например,

модуль разбора выражений должен распознавать ключевые слова Small BASIC, он не должен интерпретировать знак = как оператор, он должен выполнять оценку реляционных операторов и вычислять степени целых чисел. Именно по этим причинам функция **get_token()** и претерпела существенные изменения в связи с необходимостью удовлетворения новым расширенным требованиям к ней.

Поскольку Small BASIC использует те же самые методы, что и ранее обсуждавшийся модуль разбора выражений, вы без труда сможете понять, как он работает. Тем не менее, краткое описание модификаций, внесенных в раннюю версию программы с целью обеспечить ее работу в составе интерпретатора BASIC, будет полезным подспорьем. Начнем обсуждение с того, что дадим точное определение выражению и его соотношению с "языком" Small BASIC.

Выражения Small BASIC

Применительно к интерпретатору Small BASIC, разработанному в этой главе, выражения состоят из следующих элементов:

- Числа
- Операторы (+,–,/,*,^,=,(),<,>,>=,,+,<>)
- Переменные

В BASIC оператор ^ обозначает возведение в степень. Знак = используется как для присваиваний, так и для обозначения равенства. Однако, в отношении выражений BASIC он является оператором только при использовании в реляционных выражениях. (В стандартном BASIC присваивание является утверждением, а не операцией.) Знак <> обозначает неравенство. Все эти элементы можно комбинировать в выражениях в соответствии с правилами алгебры. Ниже приведен ряд примеров:

```
7 - 8
(100-5)*14/6
a + b - c
10^5
```

Приоритеты операций показаны в нижеприведенной таблице:

Наивысший	( )
	унарные + -
	^
	* /
	+ -
Наинизший	<> <= >= <> =

Операторы с одинаковыми приоритетами выполняются в порядке следования слева направо.

Для Small BASIC сделаны следующие допущения:

- Все переменные представляют собой одиночные буквы; это означает, что для использования доступны 26 переменных, обозначаемых буквами от A до Z. Стандартный BASIC поддерживает большее количество имен переменных. Так, например, в именах переменных стандартного BASIC за буквой могут следовать одна или несколько цифр, например, X27. Разработанный в этой главе интерпретатор Small BASIC такой возможности не поддерживает. Имена переменных из букв и цифр не реализованы в нем в интересах простоты изложения.
- Переменные не являются чувствительными к регистру (т. е., не различают строчных и прописных букв). Это означает, что для представленного здесь интерпретатора a и A будут трактоваться как одна и та же переменная.
- Все числа являются целыми, хотя вы можете с легкостью дописать процедуры для обработки других типов значений, например, таких, как значения с плавающей точкой.
- Наконец, не будут поддерживаться строковые переменные, хотя для вывода сообщений на экран могут использоваться строковые константы, заключенные в кавычки.

Все эти допущения будут встроены в модуль разбора выражений.

Элементы Small BASIC

Ядром модуля разбора выражений интерпретатора Small BASIC является функция **get_token()**. Эта функция представляет собой расширенную версию функции **get_token()**, приведенной в главе 3. Внесенные дополнения позволяют функции не только разбивать выражения на элементы, но и распознавать такие элементы языка, как ключевые слова и строки.

В Small BASIC каждый элемент имеет два формата: внешний и внутренний. Внешний формат представляет собой строковую форму, которую вы используете при написании программы. Например, "**PRINT**" будет представлять собой внешнюю форму команды **PRINT**. Хотя интерпретатор можно разработать таким образом, чтобы каждый элемент использовался во внешнем строковом формате, на практике это делается редко (если делается вообще) по причине ужасающей неэффективности такого подхода. Вместо этого используется внутренний формат элементов, представляющий собой просто целое число. Например, команда **PRINT** может быть представлена числом 1, команда **INPUT** - числом 2, и т. д. Преимущество внутреннего представления заключается в том, что при использовании целых можно написать более быстрые процедуры, чем при использовании строк. Функция **get_token()** должна преобразовать элемент из внешнего

формата во внутренний. Имейте в виду, что не все элементы будут иметь различные форматы. Например, преобразование большинства операторов просто не имеет смысла, поскольку они и так представляют собой одиночные символы.

Важно знать тип возвращаемого элемента. Например, модуль разбора выражений должен знать, является ли следующий элемент числом, оператором или переменной. В процессе разработки интерпретатора важность этого момента будет для вас очевидна.

Small BASIC хранит интерпретируемую программу в виде одной длинной строки, завершающейся нулем. Функция **get_token()** читает эту программу, считывая по одному символу за раз. Символ, который должен быть считан следующим, находится по глобальному указателю на символ. В приведенной здесь версии интерпретатора этот указатель называется **prog**. Причина определения **prog** как глобального указателя заключается в том, что он должен сохранять и поддерживать свое значение в промежутках между вызовами функции **get_token()** и позволять другим функциям получать доступ к нему. Программа разбора выражений, представленная в этой главе, использует шесть основных типов элемента: **DELIMITER** (разделитель), **VARIABLE** (переменная), **NUMBER** (число), **COMMAND** (команда), **STRING** (строка) и **QUOTE** (кавычка). Эти типы представляют собой пронумерованные значения, определенные в любом другом месте программы. Тип **DELIMITER** применим как к операторам, так и к скобкам. Тип **VARIABLE** используется для обозначения переменных, а **NUMBER** — для чисел. Тип **COMMAND** присваивается элементу в том случае, если встретилась команда BASIC. **STRING** представляет собой временный тип, используемый внутри функции **get_token()** до тех пор, пока не будет определен настоящий тип элемента. Тип **QUOTE** предназначен для строк, заключенных в кавычки. Тип элемента содержится в глобальной переменной **token_type**. Внутреннее представление элемента помещается в глобальную переменную **tok**. Ниже приведена версия функции **get_token()**, используемая Small BASIC:

```
// Получение элемента
get_token()
{
     register char *temp;

     token_type = 0;
     tok = 0;
     temp = token;

     if(*prog == '\0') { // конец файла
          *token = 0;
          tok = FINISHED;
          return(token_type=DELIMITER);
     }
     while(is_sp_tab(*prog)) ++prog; // пропускаем пробелы и табуляторы

     if(*prog == '\r') { // crlf
          ++prog; ++prog;
```

```
    tok = EOL; *token = '\r';
    token[1] = '\n'; token[2] = 0;
    return (token_type = DELIMITER);

    if(strchr("<>", *prog)) { // проверка двойной операции
        switch(*prog) {
            case '<':
                if(*(prog+1)=='>') {
                    prog++; prog++;
                    *temp = NE;
                }
                else if(*(prog+1)) == '=') {
                    prog++; prog++;
                    *temp = LE;
                else {
                    prog++;
                    *temp = '<';
                }
                temp++;
                *temp = '\0';
                break;
            case '>':
                if(*(prog+1) == '=') {
                    prog++; prog++;
                    *temp = GE;
                }
                else {
                    prog++;
                    *temp = '>';
                }
                temp++;
                *temp = '\0';
                break;
            }
            return(token_type = DELIMITER);
        }

    if(strchr("+-*^/+;(),", *prog)) { // разделитель
        *temp = *prog;
        prog++; // продвигаемся в следующую позицию
        temp++;
        *temp=0;
        return(token_type=DELIMITER);
    }
    if(*prog=='"') { // строка в кавычках
        prog++;
        while(*prog != '"' && *prog !='\r') *temp++ = *prog++;
```

```
            if(*prog=='\r') serror(MISS_QUOTE);
            prog++; *temp = 0;
            return(token_type=QUOTE);
        }
        if(isdigit(*prog)) { // число
            while(!isdelim(*prog)) *temp++ = *prog++;
            *temp = '\0';
            return(token_type=NUMBER);
        }
        if(isalpha(*prog)) { // переменная или команда
            while(!isdelim(*prog)) *temp++ = *prog++;
            token_type = STRING;
        }

        *temp = '\0';

        // не является ли строка командой или переменной
        if(token_type==STRING) {
            tok = look_up(token); // перевод во внутреннее представление
            if(!tok) token_type = VARIABLE;
            else token_type = COMMAND;
        }
        return token_type;
}
```

Рассмотрим внимательно функцию **token_type()**. Поскольку для улучшения удобочитаемости кода в выражения часто добавляются пробелы, необходимо исключить начальные пробелы. Это делается с помощью функции **is_sp_tab()**, которая возвращает значение **TRUE**, если ее аргументом является пробел или символ табуляции. После того, как пробелы будут пропущены, **prog** будет указывать на число, переменную, команду, символ перевода строки/возврата каретки, оператор, заключенную в кавычки строку или **NULL**, если программу завершают пробелы. Если следующим символом является возврат каретки, то переменная **tok** получает значение **EOL**, в переменной **token** сохраняется последовательность перевода строки/возврата каретки, а переменная **token_type** получает значение **DELIMITER**. В противном случае происходит проверка двойных операторов (типа <=). Функция **get_token()** преобразует эти операторы в их внутреннее представление. Значения **NE**, **GE** и **LE** определены по перечислению вне функции **get_token()**. Если следующий символ представляет собой одиночный оператор или другой тип разделителя, он возвращается в виде строки через глобальную переменную **token**, а переменная **token_type** получает значение **DELIMITER**. В противном случае выполняется проверка на наличие строки, заключенной в кавычки. Если и этот случай не имеет места, **get_token()** проверяет, не является ли следующий элемент числом, определяя, не является ли следующий символ цифрой. Если следующий символ является не цифрой, а буквой, то следующий элемент представляет собой перемен-

ную или команду (например, **PRINT**). Функция **look_up()** сравнивает элемент с командами, занесенными в таблицу команд, и, найдя совпадение, возвращает соответствующее внутреннее представление команды. (Функция **look_up()** будет обсуждаться далее.) Если совпадение не найдено, элемент будет считаться переменной. Наконец, если следующий символ не является ничем из вышеперечисленного, программа будет считать, что достигнут конец выражения, и переменной **token** присваивается значение NULL.

Для того, чтобы лучше понять принципы действия этой версии **get_token()**, давайте изучим, что она возвращает на каждом шаге разбора следующего выражения:

PRINT A +100 -(B*C)/2

Имейте в виду, что **token** всегда будет содержать строку, заканчивающуюся нулем, даже если она состоит из единственного символа.

Token	Token Type
PRINT	COMMAND
A	VARIABLE
+	DELIMITER
100	NUMBER
–	DELIMITER
(	DELIMITER
B	VARIABLE
*	DELIMITER
C	VARIABLE
)	DELIMITER
/	DELIMITER
2	NUMBER

Программа разбора выражений Small BASIC

Ниже приведен полный текст программы разбора выражений, модифицированной для использования с интерпретатором Small BASIC. Этот код рекомендуется поместить в отдельный файл. (Если объединить интерпретатор и программу разбора выражений в одном файле, он получится очень большим, поэтому рекомендуется использовать два отдельно скомпилированных файла.) Краткие описания значения и использования внешних переменных будут приведены после описания интерпретатора.

```
/* Small BASIC Expression parser
     Эта программа является модифицированной версией программы
     разбора выражений, обсуждавшейся в главе 3. Она предназначена
     для поддержки интерпретаторов простых языков типа Small BASIC
*/

#include <iostream.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

enum tok_types {DELIMITER, VARIABLE, NUMBER, COMMAND, STRING, QUOTE};

enum tokens {PRINT=1, INPUT, IF, THEN, FOR, NEXT, TO,
          GOTO, GOSUB, RETURN, EOL, FINISHED, END};

enum double_ops {LE=1, GE, NE};

extern char *prog; // указатель позиции в программе
extern char *p_buf; // указатель на начало программы

extern int variables[26]; // переменные

extern struct commands {
     char command[20];
     char tok;
     } table[];

extern char token[80]; //  содержит строковое представление элемента
extern char token_type; // содержит тип элемента
extern char tok; // содержит внутреннее представление элемента

void eval_exp(int &result);
void eval_exp1(int &result);
void eval_exp2(int &result);
void eval_exp3(int &result);
void eval_exp4(int &result);
void eval_exp5(int &result);
void eval_exp6(int &result);
void atom(int &result);
void get_token();
void putback();
void serror(int error);
int  get_token();
int  look_up(char *s);
int  isdelim(char c);
int  is_sp_tab(char c);
```

```
int find_var(char *s);

/* Константы, используемые для вызова serror() при синтакси-
ческих ошибках */

enum error_msg
     {SYNTAX, UNBAL_PARENS, NO_EXP, NOT_VAR, LAB_TAB_FULL, DUP_LAB
     UNDEF_LAB, THEN_EXP, TO_EXP, TOO_MNY_FOR, NEXT_WO_FOR, TOO_MNY_GOSUB,
     RET_WO_GOSUB, MISS_QUOTE};

// Точка входа
void eval_exp(int &result)
{
     get_token();
     if(!*token) {
          serror(NO_EXP); // нет выражения
          return;
     }
     eval_exp1(result);
     putback(); // возвратить последний считанный элемент во вводной поток
}

// Обработка реляционных операторов
void eval_exp1(int &result)
{
     // Реляционные операторы
     char relops[] = {
          GE, NE, LE, '<', '>', '+', 0
     };

     int temp;
     register char op;

     eval_exp2(result);
     op = *token;
     if(strchr(relops, op)) {
          get_token();
          eval_exp1(temp);
          switch(op) { // выполнение реляционных операций
               case '<':
                    result = result < temp;
                    break;
               case LE:
                    result = result <= temp;
                    break;
               case '>':
```

```
                    result = result > temp;
                    break;
                case GE:
                    result = result >= temp;
                    break;
                case '=':
                    result = result == temp;
                    break;
                case NE:
                    result = result != temp;
                    break;
            }
        }
    }

// Сложение и вычитание двух элементов
void eval_exp2(int &result);
{
    register char op;
    int temp;

    eval_exp3(result);
    while((op = *token) == '+' || op == '-') {
        get_token();
        eval_exp3(temp);
            switch(op) {
                case '-':
                    result = result - temp;
                    break;
                case '+':
                    result = result + temp;
                    break;
            }
        }
}
// Умножение и деление
void eval_exp3(int &result)
{
    register char op;
    int temp;

    eval_exp4(result);
    while((op = *token) == '*' || op == '/') {
        get_token();
        eval_exp4(temp);
        switch(op) {
```

```
                case '*':
                     result = result*temp;
                     break;
                case '/':
                     result = result/temp;
                     break;
                }
           }
}
// возведение в целую степень
void eval_exp4(int &result)
{
     int temp, ex;
     register int t;

     eval_exp5(result);
     if(*token=='^') {
          get_token();
          eval_exp4(temp);
          if(!temp) {
               result = 1;
               return;
          }
          ex = result;
          for(t=temp-1; t>0; —t) result = result*ex;
     )
}

// Унарный + или -
void eval_exp5(int &result)
{
     register char op;

     op = 0;
     if(token_type == DELIMITER) && *token== '+' || *token == '-')
     {
          op = *token;
          get_token();
     }
     eval_exp6(result);
     if(op == '-') result = -result;
}
// Обработка выражения в скобках
void eval_exp6(int &result)
{
     if(*token == '(') {
```

```
            get_token();
            eval_exp2(result);
            if(*token != ')')
                serror(UNBAL_PARENS);
            get_token();
        }
        else
            atom(result);
}

// Получение значения числа или переменной
void atom(int &result)
{
        switch(token_type) {
            case VARIABLE:
                result = find_var(token);
                get_token();
                return;
            case NUMBER:
                result = atoi(token);
                get_token();
                return;
            default:
                serror(SYNTAX);
        }
}

// Найти значение переменной
int find_var(char *s)
{
        if(!isalpha(*s)) {
            serror(NOT_VAR); // не переменная
            return 0;
        }
        return variables[toupper(*token) - 'A'];
}

// Отображение синтаксической ошибки
void serror(int error)
{
        char *p;, *temp;
        int linecount = 0;
        register int i;
        static char *e[]= {
            "Синтаксическая ошибка",
```

```
        "Незакрытые скобки",
        "Нет выражения для разбора",
        "Требуется знак равенства",
        "Не переменная",
        "Переполнена таблица меток",
        "Дублирующаяся метка",
        "Неопределенная метка",
        "Требуется THEN",
        "Требуется TO",
        "Слишком много вложенных циклов FOR",
        "NEXT требует FOR",
        "Слишком много вложений GOSUB",
        "RETURN без GOSUB",
        "Требуются двойные кавычки"
    };
    cout << e[error];

    p = p_buf;
    while( p != prog) { // найти номер строки, содержащей ошибку
        p++;
        if(*p == '\r') {
            linecount++;
        }
    }
    cout << "в строке " << linecount << ".\n";

    temp = p; // отобразить строку с ошибкой
        for(i=0; i<20; && p>p_buf && *p!='\n'; i++, p—);
        for(; p<=temp; p++) cout << *p;

    throw(1);
}

// Получение следующего элемента
get_token()
{
    register char *temp;

    token_type = 0;
    tok = 0;
    temp = token;
    if(*prog == '\0') { // конец файла;
        *token = 0;
        tok = FINISHED;
        return(token_type=DELIMITER);
    }
```

```
    while(is_sp_tab(*prog)) ++prog; // пропускаем пробелы

    if(*prog=='\r') { //crlf
    ++prog; ++prog;
    tok = EOL; *token = '\r';
    token[1] = '\n'; token[2] = 0;
    return(token_type = DELIMITER);
}

if(strchr("<>", *prog)) { // проверка двойных операторов
    switch(*prog) {
        case '<':
            if(*(prog+1)=='>') {
                prog++; prog++;
                *temp = NE;
            }
            else if(*(prog+1)=='=') {
                prog++; prog++;
                *temp = LE;
            }
            else {
                prog++;
                *temp = '<';
            }
            temp++;
            *temp = '\0';
            break;
        case '>':
            if(*(prog+1)=='=') {
                prog++; prog++;
                *temp = GE;
            }
            else {
                prog++;
                *temp = '>';
            }
            temp++;
            *temp = '\0';
            break;
        }
        return(token_type = DELIMITER);
    }
    if(strchr("+-*^/=;(),",*prog)) { // разделитель
        *temp = *prog;
        prog++;
        temp++;
        *temp = 0;
```

```
            return(token_type=DELIMITER);
      }

      if(*prog=='"') {
            prog++;
            while(*prog!='"' && *prog != '\r') *temp++ =
*prog++;
            if(*prog == '\r') serror(MISS_QUOTE);
            prog++; *temp = 0;
            return(token_type=QUOTE);
      }

      if(isdigit(*prog)) {
            while(!isdelim(*prog)) *temp++ = *prog++;
            *temp = '\0';
            return(token_type=NUMBER);
      }
      if(isalpha(*prog)) {
            while(!isdelim(*prog)) *temp++ = *prog++;
            token_type = STRING;
      }

      *temp = '\0';

// Не является ли строка командой или переменной
if(token_type == STRING) {
      tok = Look_up(token); // преобразование во внутреннее представление
      if(!tok) token_type = VARIABLE;
      else token_type = COMMAND;
      }
      return token_type;
}

// Возвратить элемент во вводной поток
void putback()
{
      char *t;

      t = token;
      for(; *t; t++) prog—;
}
// Поиск внутреннего представления элемента в таблице
look_up(char *s)
{
      register int i;
      char *p;

// Преобразование в нижний регистр
```

```
    p = s;
    while(*p) {
        *p = tolower(*p);
        p++;
    }

    for(i=0; *table[i].command; i++)
        if(!strcmp(table[i].command, s))
            return table[i].tok;
        return 0; // неизвестная команда
    }
// Возвратить TRUE, если с представляет собой разделитель
isdelim(char c)
{
    if(strchr(" ; ,+=<>/*%^=()", c) || c==9 || c=='\r' ||
c==0)
        return 1;
    return 0;

// Возвратить 1, если с - пробел или табулятор
is_sp_tab(char c)
{
    if(c==' ' || c=='\t') return 1;
    else return 0;
}
```

Приведенная здесь программа разбора может обрабатывать следующие операторы: +, -, *,/, возведение в целую степень (^) и унарный минус. Кроме того, она будет корректно обрабатывать выражения в скобках. Обратите внимание на то, что она имеет шесть уровней, как и функция **atom()**, которая возвращает значение числа. Кроме того, включены различные процедуры поддержки и код функции **get_token()**.

Для оценки выражения необходимо установить указатель **prog** таким образом, чтобы он указывал на начало строки, в которой содержится выражение, после чего вызвать функцию **eval_exp()**, указав ей аргументом переменную, в которую требуется поместить результат. Обратите внимание на отличия функции **eval_exp1()** от аналогичной функции, обсуждавшейся в главе 3, где эта функция обрабатывала оператор присваивания. Однако, в BASIC присваивание является утверждением, а не оператором. Поэтому функция **eval_exp1()** и не используется в этих целях при разборе выражений, обнаруженных в программах BASIC. Вместо этого функция используется для обработки реляционных операторов. Однако, если вы используете интерпретатор для экспериментов с другими языками, вам может потребоваться добавление функции **eval_exp0()**, которая будет обрабатывать присваивание как оператор.

Особое внимание следует обратить на функцию **serror()**, использующуюся для вывода сообщений об ошибках. При выявлении синтаксической ошибки вызывается функция **serror()** с аргументом, соответствующим номеру обнаруженной

ошибки. После этого функция выводит соответствующее сообщение об ошибке, номер строки, в которой эта ошибка обнаружена, а также фрагмент строки, содержащий эту ошибку. Наиболее рационально вызывать функцию **serror()**, используя пронумерованные значения, перечисленные в **error_msg** в самом начале кода программы разбора выражений. Фактически, сообщение "Синтаксическая ошибка" используется только в случае неприменимости ни одного из предусмотренных случаев. Во всех остальных ситуациях выводится конкретизированное сообщение об ошибке. Обратите внимание на то, что функция **serror()** завершается утверждением обработки исключений **throw**. Это исключение должно быть отслежено утверждением **catch**, которое бы предпринимало некоторые разумные действия. В целях интерпретатора Small BASIC утверждение **catch** помещено в функцию **main()**. Оно просто прерывает выполнение программы.

Примечание: Если ваш компилятор не поддерживает ключевых слов обработки исключений С++ (**try, catch** и **throw**), вы должны будете использовать для этой цели функции языка С (**setjmp()** и **longjmp()**).

Как программа разбора выражений обрабатывает переменные

Необходимо дать хотя бы краткое объяснение принципов, в соответствии с которыми программа разбора выражений обрабатывает переменные. Как уже упоминалось ранее, интерпретатор Small BASIC будет распознавать в качестве переменных только алфавитные символы A — Z. Каждая переменная занимает одну позицию в 26-элементном массиве целых, называемых переменными. В коде интерпретатора этот массив определяется следующим образом, причем каждая переменная инициализируется нулем:

```
int variables[26] = { // 26 переменных, A - Z
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0
};
```

Поскольку в качестве имен переменных используются буквы A — Z, их можно использовать для индексации массива переменных. Для этого достаточно вычесть из имени переменной ASCII-код символа 'A'. Значение переменной находит функция **find_var()**. Эта функция содержится в полном листинге программы разбора выражений, а здесь приведена только для вашего удобства.

```
int find_var(char *s)
{
    if(!isalpha(*s)) {
        serror(NOT_VAR); // не переменная
```

```
        return 0;
    }
    return variables[toupper(*token) - 'A'];
}
```

Обратите внимание, что в том виде, как она представлена сейчас, функция **find_var()** фактически будет воспринимать и длинные имена переменных, но фактически значимой будет только первая буква. При желании вы можете модифицировать эту функцию так, чтобы длинные имена переменных считались ошибкой.

Интерпретатор Small BASIC

Фактически все интерпретаторы состоят из двух основных частей — программы разбора выражений (**parser**) и собственно интерпретатора, который осуществляет выполнение программы. Этот раздел посвящен изучению модуля интерпретатора.

Ключевые слова

Как уже говорилось в начале этой главы, Small BASIC интерпретирует небольшое подмножество языка BASIC. Ниже приведен список ключевых слов, распознаваемых этим интерпретатором:

PRINT

INPUT

IF

THEN

FOR

NEXT

TO

GOTO

GOSUB

RETURN

END

Внутреннее представление этих ключевых слов, а также символов конца строки (**EOL**) и конца программы (**FINISHED**) показано ниже:

```
enum tokens { PRINT = 1, INPUT, IF, THEN, FOR, NEXT, TO,
              GOTO, GOSUB, RETURN, EOL, FINISHED, END};
```

Нумерация ключевых слов начинается с 1, так как значение 0 используется функцией **look_up()** для обозначение неизвестной команды.

Для того, чтобы внешнее представление команды могло быть преобразовано во внутреннее, как внешний, так и внутренний форматы помещаются в массив структур, называемый таблицей. Таблица для интерпретатора Small BASIC приведена ниже:

```
// Таблица поиска ключевых слов
struct commands {
     char command[20]; // строковая форма
     char tok; // внутреннее представление
     } table[] = {
     "print", PRINT,
     "input", INPUT,
     "if", IF,
     "then", THEN,
     "for", FOR,
     "next", NEXT,
     "to", TO,
     "goto", GOTO,
     "gosub", GOSUB,
     "return",RETURN,
     "end", END,
     "", END // отмечает конец таблицы
};
```

Обратите внимание на то, что нулевая строка помечает конец таблицы. Показанная здесь функция **look_up()** использует эту таблицу для поиска и возвращает или внутреннее представление элемента, или нуль, если совпадение не найдено. (Эта функция является частью приведенного ранее файла модуля разбора выражений.)

```
// Поиск внутреннего представления элемента в таблице
look_up(char *s)
{
     register int i;
     char *p;
// Преобразование в нижний регистр
     p = s;
     while(*p) {
```

```
        *p = tolower(*p);
        p++;
    }

    for(i=0; *table[i].command; i++)
        if(!strcmp(table[i].command, s))
            return table[i].tok;
        return 0; // неизвестная команда
    }
```

Загрузка программы

В состав интерпретатора Small BASIC не включено целостного редактора. Программу на BASIC можно писать, используя любой стандартный текстовый редактор. При запуске интерпретатора Small BASIC он считывает и исполняет программу. Код функции (она называется **load_program( )**), выполняющей загрузку программы, приведен ниже:

```
// Загрузка программы
load_program(char *p, char *fname)
{
    ifstream in(fname, ios::in | ios::binary);
    int i=0;

    if(!in) {
        cout << "File not found";
        cout << "- be sure to specify .BAS extension.\n";
        return 0;
    }

    i = 0;
    do {
        *p = in.get();
        p++; i++;
    } while(!in.eof() && i<PROG_SIZE);

    // null terminate the program
    if(*(p-2)==0x1a) *(p-2) = '\0'; // discard eof marker
    else *(p-1) = '\0';

    in.close();
    return 1;
}
```

Эта функция отбросит завершающий программу маркер конца файла EOF, если он будет найден в файле. Как вы знаете, некоторые редакторы добавляют этот маркер в конец текстового файла, а некоторые — не добавляют. Функция **load_program()** предусматривает оба случая.

Главный цикл

Все интерпретаторы управляются главным циклом (верхний уровень). Этот цикл работает по следующему принципу: он считывает следующий элемент программы, а затем выбирает соответствующее действие, необходимое для его обработки. Интерпретатор Small BASIC — не исключение из этого правила. Его главный цикл выглядит следующим образом:

```
do {
     token_type = get_token();
     // проверка на утверждение присваивания
     if(token_type = VARIABLE) {
          putback(); // возвращаем переменную во вводной по-
ток
          assignment(); // должно быть утверждение присваива-
ния
     }
     else // команда
          switch(tok) {
               case PRINT:
                    print();
                    break;
               case GOTO:
                    exec_goto();
                    break;
               case IF:
                    exec_if();
                    break;
               case FOR:
                    exec_for();
                    break;
               case NEXT:
                    next();
                    break;
               case INPUT:
                    input();
                    break;
               case GOSUB:
                    gosub();
                    break;
               case RETURN:
```

```
                greturn();
                break;
            case END:
                return 0;
            }
    } while (tok != FINISHED);
```

Сначала из программы считывается элемент. В языке BASIC первый элемент строки определяет тип утверждения, содержащегося в данной строке. (На этом шаге никакого заглядывания вперед не требуется.) Если предположить отсутствие синтаксических ошибок, то утверждение будет утверждением присваивания, если первый элемент является переменной. Если это не так, то элемент должен быть командой, и в этом случае на основании значения переменной **tok** будет выбрано соответствующее утверждение **case**. Далее мы рассмотрим, как работает каждая из этих команд.

Функция присваивания

Как уже упоминалось ранее, в языке BASIC присваивание является утверждением, а не операцией. Общая форма утверждения присваивания в BASIC выглядит следующим образом:

var_name = expression

Утверждение присваивания интерпретируется приведенной ниже функцией **assignment():**

```
// Присваивание значения переменной
void assignment()
{
    int var, value;

    // получение имени переменной
    get_token();
    if(!isalpha(*token)) {
        serror(NOT_VAR);
        return;
    }

    // преобразование в индекс таблицы переменных
    var = toupper(*token) - 'A';
    // получение знака равенства
    get_token();
    if(*token != '=') {
```

```
            serror(EQUAL_EXP);
            return;
        }

        // получение значения для присваивания
        eval_exp(value);

        // присваивание значения
        variables[var] = value;
}
```

Первая строка функции **assignment()** считывает элемент из программы. Этот элемент и будет переменной, которой будет присвоено значение. Если этот элемент не является допустимой переменной, выводится сообщение об ошибке. Далее происходит считывание знака равенства. Затем вызывается функция **eval_exp()**, которая вычисляет значение, присваиваемое переменной. Наконец, вычисленное значение присваивается переменной. Функция удивительно проста и логична, вследствие того, что большая часть запутанной и кропотливой работы выполняется программой разбора выражений и функцией **get_token()**.

Команда PRINT

В языке BASIC функция PRINT является мощной и очень гибкой. Хотя разработка полнофункциональной модели PRINT и выходит за рамки данной главы, все же представленная здесь функция поддерживает наиболее важные из встроенных возможностей команды PRINT. В BASIC общая форма команды PRINT выглядит следующим образом:

PRINT *arg-list*

где *arg-list* представляет собой список выражений или заключенных в кавычки строк, разделителем между которыми служит запятая или точка с запятой. Нижеприведенная функция **print()** интерпретирует команду PRINT:

```
// Простая версия команды PRINT языка BASIC
void print()
{
        int result;
        int len = 0; spaces;
        char last_delim, str[80];

        do {
                get_token(); // получаем следующий элемент
                if(tok == EOL || tok == FINISHED) break;
                if(token_type == QUOTE) { // строка
```

```
                cout << token;
                len += strlen(token);
                get_token();
            }
            else { // выражение
                putback();
                eval_exp(result);
                get_token();
                cout << result;
                itoa(result, str, 10);
                len += strlen(str); // сохраняем длину
            }
            last_delim = *token;
            // если запятая, переходим в следующую позицию табулятора
            if(*token == ',') {
                // вычисляем количество позиций для перемещения
                spaces = 8 - (len%8);
                len += spaces;
                while(spaces) {
                    cout << @ @;
                    spaces —;
                }
            }
            else if(*token== ';') {
                cout << @ @;
                len++;
            }
            else if(tok != EOL && tok != FINISHED)
serror(SYNTAX);
        } while (*token == ';' || *token == ',');

        if(tok == EOL || tok == FINISHED) {
            if(last_delim != ';' && last_delim != ',')
                cout << endl;
        }
        else serror(SYNTAX);
}
```

Команда PRINT может использоваться для вывода на экран списка переменных и заключенных в кавычки строк. Если элементы отделены друг от друга двоеточием, то между ними вставляется пробел. Если в качестве разделителя символов использовалась запятая, то следующий элемент будет выведен, начиная со следующей позиции табулятора. Если список завершается запятой или двоеточием, то перевода строки не осуществляется. Ниже приведен ряд примеров правильного использования команды PRINT:

PRINT X; Y; "THIS IS A STRING"

PRINT 10/4

PRINT

В результате выполнения последнего примера на экран выводится пустая строка.

Обратите внимание на то, что функция **print()** использует функцию **putback()** для возврата элемента во вводной поток. Причина, по которой функция **print()** должна заглядывать вперед, заключается в том, что она должна определять, что собой представляет следующий печатаемый элемент — заключенную в кавычки строку или числовое выражение. Если следующий печатаемый элемент представляет собой выражение, то его первый элемент должен быть возвращен во вводной поток, с тем чтобы программа разбора выражений могла правильно вычислить его значение.

Команда INPUT

В языке BASIC команда INPUT используется для считывания информации, вводимой с клавиатуры в переменную. Эта команда имеет два общих формата:

INPUT *var-name*

и

INPUT *"prompt-string", var-name*

Первая форма команды отображает на экране знак вопроса и ожидает ввода. Вторая форма отображает приглашение ко вводу, заданное строкой "prompt-string" и также ожидает ввода. Команду INPUT реализует нижеприведенный код функции **input()**:

```
// Простая форма команды INPUT для BASIC
void input()
{
    char var;
    int i;

    get_token(); // проверяем, присутствует ли строка-приглашение
    if(token_type == QUOTE) {
        cout << token;
        get_token();
        if(*token != ',') serror(SYNTAX);
        get_token();
    }
    else cout << "? "; // в противном случае приглашением будет ?
    var = toupper(*token) - 'A'; // получаем переменную для ввода
```

```
    cin >> i; // читаем ввод

    variables[var] = i; // сохраняем ввод
}
```

Работа этой функции проста и должна быть понятна из комментариев, которыми снабжен ее текст.

Команда GOTO

Теперь, когда вы изучили принципы работы некоторых простых команд, настало время для разработки более сложных. В языке BASIC наиболее важной формой управления программой является оператор GOTO. В стандартном BASIC аргументом GOTO должен быть номер строки. Этот традиционный подход сохранен и в Small BASIC. Однако, Small BASIC не требует наличия номеров у всех строк. Номер строки должен присутствовать только для тех строк, которые являются конечной целью перехода с помощью операторах GOTO. Общая форма оператора GOTO выглядит следующим образом:

GOTO *line-number*

Основная сложность, связанная с реализацией GOTO, заключается в том, что необходимо разрешить переходы как вперед, так и назад по тексту программы. Для того, чтобы эффективно реализовать это требование, необходимо перед исполнением просканировать всю программу и поместить в специальную таблицу точное местоположение всех меток. Таким образом, при каждом вызове GOTO местоположение нужной строки может быть найдено из этой таблицы, после чего управление будет передано в эту точку. Таблица, содержащая метки, декларируется следующим образом:

```
// Таблица для поиска меток
struct label {
    char name[LAB_LEN]; //метка
    char *p; // указатель на позицию метки в исходном файле
    } label_table[NUM_LAB];
```

Процедура, сканирующая программу и помещающая в эту таблицу все встретившиеся метки, называется **scan_labels()**. Ее текст вместе со вспомогательными функциями приведен ниже.

```
// Находим все метки
void scan_labels()
{
    int addr;
    char *temp;
```

```
    label_init(); //обнуляем все метки
    temp = prog; // сохраняем указатель на начало программы

    // если первый элемент файла является меткой
    get_token();
    if(token_type == NUMBER) {
        strcpy(label_table[0].name, token);
        label_table[0].p = prog;
    }

    find_eol();
    do {
        get_token();
        if(token_type == NUMBER) {
            addr = get_next_label(token);
            if(addr == -1 || addr == -2) {
                (addr == -1) ?
serror(LAB_TAB_FULL):serror(DUP_LAB);
            }
            strcpy(label_table[addr].name, token);
            label_table[addr].p = prog;
        }

        // если не находимся в пустой строке, найти следую-
щую строку
        if(tok != EOL) find_eol();
        } while (tok != FINISHED);
        prog = temp; // восстанавливаем исходное положение
}

// Находим начало следующей строки
void find_eol()
{
    while(*prog != '\n' && *prog != '\0') ++prog;
    if(*prog) prog++;
}

/* Эта функция возвращает индекс следующей свободной позиции
в массиве меток.
    Возвращает -1, если массив заполнен
    Возвращает -2, если находит дублирующуюся метку
*/
get_next_label(char *s)
{
    register int i;
```

```
    for (i=0; i<NUM_LAB; ++i) {
        if(label_table[i].name[0] == 0) return i;
        if(!strcmp(label_table[i].name, s)) return -2;
    }
    return -1;
}

/* Эта функция находит местоположение заданной метки. Если
метка не найдена, функция возвращает NULL; в противном случае
она возвращает указатель на позицию метки
*/
char *find_label(char *s)
{
    register int i;

    for(i=0; i<NUM_LAB; ++i)
        if(!strcmp(label_table[i].name, s)) return
label_table[i].p;
    return '\0';
}

/* Эта функция инициализирует массив, содержащий метки. Нуле-
вое имя метки означает неиспользованную позицию в массиве
*/
void label_init()
{
    register int i;

    for(i=0; i<NUM_LAB; ++i)
        label_table[i].name[0] = '\0';
}
```

Функция **scan_labels()** сообщает об ошибках двух типов. Первый тип ошибки — это дублирующиеся метки. BASIC (и большинство других языков программирования) не допускает наличия в программе двух одинаковых меток. Ошибки второго типа возникают при переполнении таблицы меток. Размер этой таблицы определяется переменной **NUM_LAB**, для которой вы можете задать любой размер.

После построения таблицы меток выполнять инструкции GOTO будет достаточно легко. Эту команду выполняет нижеприведенная функция **exec_goto()**:

```
// Выполнение GOTO
Execute a GOTO statement.
void exec_goto()
```

```
{
     char *loc;
     get_token(); // получить метку, к которой требуется пе-
рейти
     // найти местоположение метки
     loc = find_label(token);
     if(loc==NULL)
          serror(UNDEF_LAB); // метка не определена
     else prog = loc; // запустить программу, начиная с этой
метки
}
```

Вспомогательная функция **find_label()** ищет метку в таблице и возвращает указатель на нее. Если метка не найдена, функция возвращает NULL, значение, которое никогда не может представлять собой допустимый указатель. Если адрес не является нулевым, он присваивается переменной **prog**, что вызывает возобновление выполнения программы, начиная с точки местоположения метки. (Не забывайте, что **prog** представляет собой указатель на точку, в которой на текущий момент выполняется программа.) Если метка не найдена, выводится диагностическое сообщение о неопределенной метке.

Утверждение IF

Интерпретатор Small BASIC выполняет утверждение IF, представляющее собой подмножество утверждения IF стандартного BASIC. В Small BASIC не допускается ключевое слово ELSE. (Однако, поняв принцип действия IF, вы сами с легкостью добавите ELSE в интерпретатор Small BASIC.) Общая форма утверждения IF выглядит следующим образом:

IF *expression rel-op expression* THEN *statement*

Утверждение, следующее за THEN, выполняется только в том случае, если в результате выполнения реляционного оператора получается значение TRUE. Утверждение IF выполняет функция **exec_if()**, текст которой приведен ниже:

```
// Выполнение утверждения IF.
void exec_if()
{
  int result;
  char op;

  eval_exp(result); // получаем значение выражения

  if(result) { //
    get_token();
    if(tok!=THEN) {
```

```
        serror(THEN_EXP);
        return;
      } //
    }
    else find_eol(); // найти начало следующей строки
}
```

Функция **exec_if()** работает следующим образом. Сначала вычисляется значение реляционного выражения. Если значение истинно, выполняется выражение THEN, в противном случае функция **find_eol()** находит начало следующей строки. Обратите внимание, что если выражение получает значение TRUE, то функция **exec_if()** просто возвращает управление. Это приводит к итерации в главном цикле, в результате которой считывается следующий элемент. Возврат в главный цикл приводит к простому выполнению конечного утверждения, точно так же, как это происходило бы, если бы оно было расположено в следующей строке. Если выражение имеет значение FALSE, то команда находит начало следующей строки, после чего управление возвращается в главный цикл.

Цикл FOR

Решение проблемы с циклом FOR для интерпретатора Small BASIC представляет собой довольно трудную задачу, которая, однако, имеет очень элегантное решение. Общая форма цикла FOR выглядит следующим образом:

FOR *control-var* = *initial-value* TO *target-value*

.

NEXT

Версия цикла FOR для Small BASIC допускает только циклы с положительным приращением управляющей переменной, причем с каждой итерацией это приращение равно 1. Команда STEP не поддерживается.

В BASIC, как и в C++, циклы могут быть вложенными на нескольких уровнях. Это представляет основную проблему, так как каждое ключевое слово NEXT должно ассоциироваться с конкретным ключевым словом FOR. Решение этой проблемы заключается в реализации цикла FOR на базе механизма стеков. В начале цикла вся информация о состоянии управляющей переменной, конечном значении, а также местоположении начала цикла в программе помещается в стек. Каждый раз, когда встречается ключевое слово NEXT, происходит извлечение информации из стека, управляющая переменная модифицируется, после чего ее значение сравнивается с конечным значением. Если управляющая переменная имеет значение, которое больше конечного, цикл останавливается, а выполнение программы продолжается со строки, следующей за ключевым словом NEXT. В

противном случае модифицированная информация снова помещается в стек, и выполнение цикла продолжается. Такая реализация цикла FOR подходит не только для одиночных циклов, но и для вложенных. поскольку в этом случае самое глубоко вложенное NEXT всегда будет ассоциироваться с самым глубоко вложенным FOR. (Информация, помещенная в стек последней, извлекается из него первой.) После завершения внутреннего цикла его информация извлекается из стека. Если существовал внешний цикл, то на вершину стека попадает именно его информация. Таким образом, каждое ключевое слово NEXT автоматически ассоциируется с соответствующим ключевым словом FOR.

Для поддержки цикла FOR необходимо создать стек, который содержал бы информацию о циклах. Пример его реализации приведен ниже:

```
// поддержка для циклов FOR
struct for_stack {
  int var; // переменная-счетчик
  int target; // конечное значение
  char *loc; // местоположение начала цикла в исходном коде
} fstack[FOR_NEST]; // стек для цикла FOR/NEXT
```

Значение FOR_NEST определяет, насколько глубоко могут быть вложены циклы FOR. (Как правило, двадцати пяти бывает более, чем достаточно.)

Стек FOR управляется двумя функциями стека, называющимися **fpush()** и **fpop()**. Тексты этих функций приведены ниже:

```
// Push the FOR stack.
void fpush(struct for_stack stckvar)
{
  if(ftos==FOR_NEST)
    serror(TOO_MNY_FOR);

  fstack[ftos] = stckvar;
  ftos++;
}

// Pop the FOR stack.
struct for_stack fpop()
{
  if(ftos==0)
    serror(NEXT_WO_FOR);
  ftos—;
  return(fstack[ftos]);
}
```

Теперь, обеспечив всю необходимую поддержку и разработав вспомогательные функции, можно приступить к разработке функций, выполняющих утверждения FOR и NEXT. Эти функции приведены ниже:

```
// Execute a FOR loop.
void exec_for()
{
  struct for_stack stckvar;
  int value;

  get_token(); // read the control variable
  if(!isalpha(*token)) {
    serror(NOT_VAR);
    return;
  }
  // save index of control var
  stckvar.var = toupper(*token)-'A';

  get_token(); // read the equal sign
  if(*token != '=') {
    serror(EQUAL_EXP);
    return;
  }

  eval_exp(value); // get initial value

  variables[stckvar.var] = value;

  get_token();
  if(tok!=TO) serror(TO_EXP); // read and discard the TO

  eval_exp(stckvar.target); // get target value

  /* if loop can execute at least once,
     push info on stack */
  if(value >= variables[stckvar.var]) {
    stckvar.loc = prog;
    fpush(stckvar);
  }
  else // otherwise, skip loop code altogether
    while(tok!=NEXT) get_token();
}
// Execute a NEXT statement.
void next()
{
  struct for_stack stckvar;
```

```
  stckvar = fpop(); // read the loop info

  variables[stckvar.var]++; // increment control var

  // if done, return
  if(variables[stckvar.var] > stckvar.target) return;

  fpush(stckvar);  // otherwise, restore the info
  prog = stckvar.loc;  // loop
}
```

Вам будет несложно понять принципы работы этих процедур, читая комментарии. Этот код, в том виде, как он здесь представлен, не предотвращает выхода GOTO за пределы цикла FOR. Однако, выход за пределы цикла FOR повредит стек FOR, поэтому таких ситуаций следует избегать.

Решение проблемы цикла FOR на базе стека можно обобщить для всех циклов. Хотя Small BASIC не реализует никаких других утверждений для циклов, вы можете добавить в него процедуры для циклов любого типа, включая WHILE и DO/WHILE. Кроме того, как будет видно из следующего раздела, решение на базе стека можно применить к любой языковой конструкции, допускающей вложение, включая вызовы подпрограмм.

Конструкция GOSUB

Хотя стандартный BASIC не поддерживает независимых подпрограмм, он позволяет осуществлять вызовы и возврат управления из отдельных частей программы с помощью утверждений GOSUB и RETURN. Общая форма утверждения GOSUB/RETURN выглядит следующим образом:

GOSUB *line-num*

.

line-num

subroutine code

RETURN

Вызов подпрограммы, даже такой ограниченной, как реализованная в BASIC, требует использования стека. Причина этого аналогична причине, по которой решение на базе стека было применено для реализации утверждения FOR. Это делается для того, чтобы стали допустимыми вложенные вызовы подпрограмм. Поскольку возможны ситуации, когда одна подпрограмма будет вызывать дру

гую, стек необходим для того, чтобы правильно ассоциировать утверждения RETURN с соответствующими им утверждениями GOSUB. Стек для GOSUB определяется следующим образом:

```
char *gstack[SUB_NEST]; // стек для gosub
```

Как видите, **gstack** представляет собой просто массив указателей на символы. Он содержит точки возврата в программу после завершения подпрограмм.

Ниже приведен листинг функции **gosub()** и ее вспомогательных процедур:

```
// Execute a GOSUB command.
void gosub()
{
  char *loc;

  get_token();

  // find the label to call
  loc = find_label(token);
  if(loc==NULL)
    serror(UNDEF_LAB); // label not defined
  else {
    gpush(prog); // save place to return to
    prog = loc;  // start program running at that loc
  }
}

// Return from GOSUB.
void greturn()
{
   prog = gpop();
}

// Push GOSUB stack.
void gpush(char *s)
{
  if(gtos==SUB_NEST)
    serror(TOO_MNY_GOSUB);
  gstack[gtos] = s;
  gtos++;
}

// Pop GOSUB stack.
char *gpop()
```

```
{
  if(gtos==0)
    serror(RET_WO_GOSUB);

  gtos--;
  return(gstack[gtos]);
}
```

Поясним принцип работы команды GOSUB. Когда интерпретатор встречает в программе утверждение GOSUB, текущее значение указателя **prog** помещается в стек GOSUB. (Это та точка, в которую будет передано управление из подпрограммы после ее завершения.) Затем программа находит номер строки назначения, и адрес, ассоциированный с ней, присваивается переменной **prog**. Выполнение программы продолжается из точки начала подпрограммы. Когда интерпретатор встречает утверждение RETURN, данные извлекаются из стека GOSUB и присваиваются указателю **prog**. Это вызывает продолжение выполнения программы со строки, следующей за утверждением GOSUB. Поскольку адрес точки возврата помещается в стек GOSUB, вызовы подпрограмм могут быть вложенными. На вершине стека **gstack** всегда будет находиться адрес возврата подпрограммы, которая была вызвана последней. Этот процесс позволяет реализовать любую глубину вложения GOSUB.

Полный код интерпретатора

Здесь приведен весь код интерпретатора Small BASIC, за исключением процедур, находящихся в файле программы разбора выражений. Введите оба файла (интерпретатор и модуль разбора выражений) на свой компьютер и скомпилируйте их. После этого постройте исполняемый файл и назовите его SBASIC.

```
/* A Small BASIC interpreter

   You can easily expand this interpreter or
   use it as a starting point for developing
   your own computer language.
*/

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
```

```
const int NUM_LAB = 100;
const int LAB_LEN = 10;
const int FOR_NEST = 25;
const int SUB_NEST = 25;
const int PROG_SIZE = 10000;

enum tok_types {DELIMITER, VARIABLE, NUMBER, COMMAND,
                STRING, QUOTE};

enum tokens {PRINT=1, INPUT, IF, THEN, FOR, NEXT, TO,
             GOTO, GOSUB, RETURN, EOL, FINISHED, END};

char *prog; // points into the program
char *p_buf; // points to start of program

int variables[26]= { // 26 user variables,  A-Z
  0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0
};
// keyword lookup table
struct commands {
  char command[20]; // string form
  char tok; // internal representation
} table[] = { // commands must be entered lowercase
  "print", PRINT, // in this table.
  "input", INPUT,
  "if", IF,
  "then", THEN,
  "goto", GOTO,
  "for", FOR,
  "next", NEXT,
  "to", TO,
  "gosub", GOSUB,
  "return", RETURN,
  "end", END,
  "", END  // mark end of table
};

char token[80];
char token_type, tok;

// label lookup table
struct label {
  char name[LAB_LEN]; // label
  char *p; // points to label's location in source file
} label_table[NUM_LAB];
```

```
// support for FOR loops
struct for_stack {
  int var; // counter variable
  int target; // target value
  char *loc; // place in source code to loop to
} fstack[FOR_NEST]; // stack for FOR/NEXT loop

char *gstack[SUB_NEST]; // stack for gosub

int ftos;  // index to top of FOR stack
int gtos;  // index to top of GOSUB stack

void print();
void scan_labels();
void find_eol();
void exec_goto();
void exec_if();
void exec_for();
void next();
void fpush(struct for_stack i);
void input();
void gosub();
void greturn();
void gpush(char *s);
void label_init();
void assignment();
char *find_label(char *s);
char *gpop();
struct for_stack fpop();
int load_program(char *p, char *fname);
int get_next_label(char *s);

// prototypes for functions in the parser file
void eval_exp(int &result);
int get_token();
void serror(int error), putback();
/* These are the constants used to call serror() when
   a syntax error occurs.  Add more if you like.
   NOTE: SYNTAX is a generic error message used when
   nothing else seems appropriate.
*/
enum error_msg
     {SYNTAX, UNBAL_PARENS, NO_EXP, EQUAL_EXP,
      NOT_VAR, LAB_TAB_FULL, DUP_LAB, UNDEF_LAB,
      THEN_EXP, TO_EXP, TOO_MNY_FOR, NEXT_WO_FOR,
      TOO_MNY_GOSUB, RET_WO_GOSUB, MISS_QUOTE};
```

```
main(int argc, char *argv[])
{
  if(argc!=2) {
    cout << "Usage: sbasic <filename>\n";
    exit(1);
  }

  // allocate memory for the program
  prog = new char [PROG_SIZE];
  if(!prog) {
    cout << "Allocation Failure";
    exit(1);
  }
  p_buf = prog;

  // load the program to execute
  if(!load_program(prog, argv[1])) exit(1);
  // begin try block
  try {
    scan_labels(); // find the labels in the program
    ftos = 0; // initialize the FOR stack index
    gtos = 0; // initialize the GOSUB stack index/
    do {
      token_type = get_token();
      // check for assignment statement
      if(token_type==VARIABLE) {
        putback(); // return the var to the input stream
        assignment(); // must be assignment statement
      }
      else // is command
        switch(tok) {
          case PRINT:
            print();
            break;
          case GOTO:
            exec_goto();
            break;
          case IF:
            exec_if();
            break;
          case FOR:
            exec_for();
            break;
          case NEXT:
            next();
            break;
          case INPUT:
```

```
            input();
            break;
          case GOSUB:
            gosub();
            break;
          case RETURN:
            greturn();
             break;
          case END:
            return 0;
        }
    } while (tok != FINISHED);
  } // end of try block

  /* catch throws here.  As implemented, only
     serror() throws an exception.  However,
     when creating your own languages, you can
     throw a variety of different exceptions.
  */
  catch(int) {
    return 0; // fatal error
  }
  return 0;
}
// Load a program.
load_program(char *p, char *fname)
{
  ifstream in(fname, ios::in | ios::binary);
  int i=0;

  if(!in) {
    cout << "File not found ";
    cout << "- be sure to specify .BAS extension.\n";
    return 0;
  }

  i = 0;
  do {
    *p = in.get();
    p++; i++;
  } while(!in.eof() && i<PROG_SIZE);

  // null terminate the program
  if(*(p-2)==0x1a) *(p-2) = '\0'; // discard eof marker
  else *(p-1) = '\0';

  in.close();
  return 1;
}
```

```
// Find all labels.
void scan_labels()
{
  int addr;
  char *temp;
  label_init(); // zero all labels
  temp = prog;  // save pointer to top of program

  // if the first token in the file is a label
  get_token();
  if(token_type==NUMBER) {
    strcpy(label_table[0].name, token);
    label_table[0].p = prog;
  }

  find_eol();
  do {
    get_token();
    if(token_type==NUMBER) {
      addr = get_next_label(token);
      if(addr == -1 || addr == -2) {
        (addr == -1) ? serror(LAB_TAB_FULL):serror(DUP_LAB);
      }
      strcpy(label_table[addr].name, token);
// save current location in program
      label_table[addr].p = prog;
    }
    // if not on a blank line, find next line
    if(tok!=EOL) find_eol();
  } while(tok!=FINISHED);
  prog = temp; // restore original location
}
// Find the start of the next line.
void find_eol()
{
  while(*prog!='\n'  && *prog!='\0') ++prog;
  if(*prog) prog++;
}

/* Return index of next free position in label array.
   -1 is returned if the array is full.
    2 is returned when duplicate label is found.
*/
get_next_label(char *s)
{
  register int i;
```

```
  for(i=0; i<NUM_LAB; ++i) {
    if(label_table[i].name[0]==0) return i;
    if(!strcmp(label_table[i].name, s)) return -2;
  }

  return -1;
}

/* Find location of given label.  A null is returned
   if label is not found; otherwise a pointer to the
   position of the label is returned.
*/
char *find_label(char *s)
{
  register int i;

  for(i=0; i<NUM_LAB; ++i)
    if(!strcmp(label_table[i].name, s))
      return label_table[i].p;
  return '\0'; // error condition
}
/* Initialize the array that holds the labels.
   By convention, a null label name indicates that
   the array position is unused.
*/
void label_init()
{
  register int i;

  for(i=0; i<NUM_LAB; ++i)
    label_table[i].name[0] = '\0';
}

// Assign a variable a value.
void assignment()
{
  int var, value;

  // get the variable name
  get_token();
  if(!isalpha(*token)) {
    serror(NOT_VAR);
    return;
  }

  // convert to index into variable table
  var = toupper(*token) 'A';
```

```
  // get the equal sign
  get_token();
  if(*token != '=') {
    serror(EQUAL_EXP);
    return;
  }

  // get the value to assign
  eval_exp(value);

  // assign the value
  variables[var] = value;
}
// Execute a simple version of the BASIC PRINT statement.
void print()
{
  int result;
  int len=0, spaces;
  char last_delim, str[80];
  do {
    get_token(); // get next list item
    if(tok==EOL || tok==FINISHED) break;
    if(token_type==QUOTE) { // is string
      cout << token;
      len += strlen(token);
      get_token();
    }
    else { // is expression
      putback();
      eval_exp(result);
      get_token();
      cout << result;
      itoa(result, str, 10);
      len += strlen(str); // save length
    }
    last_delim = *token;

    // if comma, move to next tab stop
    if(*token == ',') {
      // compute number of spaces to move to next tab
      spaces = 8 - (len % 8);
      len += spaces; // add in the tabbing position
      while(spaces) {
        cout << " ";
        spaces--;
      }
    }
```

```
    else if(*token==';') {
      cout << " ";
      len++;
    }
    else if(tok!=EOL && tok!=FINISHED) serror(SYNTAX);
  } while (*token==';' || *token==',');
 if(tok==EOL || tok==FINISHED) {
    if(last_delim != ';' && last_delim != ',')
      cout << endl;
  }
  else serror(SYNTAX);

}

// Execute a GOTO statement.
void exec_goto()
{
  char *loc;

  get_token(); // get label to go to

  // find the location of the label
  loc = find_label(token);
  if(loc==NULL)
    serror(UNDEF_LAB); // label not defined

  else prog = loc;  // start program running at that loc
}

// Execute an IF statement.
void exec_if()
{
  int result;
  char op;

  eval_exp(result); // get value of expression

  if(result) { // is true so process target of IF
    get_token();
    if(tok!=THEN) {
      serror(THEN_EXP);
      return;
    } // else, target statement will be executed
  }
  else find_eol(); // find start of next line
}
```

```
// Execute a FOR loop.
void exec_for()
{
  struct for_stack stckvar;
  int value;

  get_token(); // read the control variable
  if(!isalpha(*token)) {
    serror(NOT_VAR);
    return;
  }
  // save index of control var
  stckvar.var = toupper(*token)-'A';

  get_token(); // read the equal sign
  if(*token != '=') {
    serror(EQUAL_EXP);
    return;
  }
  eval_exp(value); // get initial value

  variables[stckvar.var] = value;

  get_token();
  if(tok!=TO) serror(TO_EXP); // read and discard the TO

  eval_exp(stckvar.target); // get target value

  /* if loop can execute at least once,
     push info on stack */
  if(value >= variables[stckvar.var]) {
    stckvar.loc = prog;
    fpush(stckvar);
  }
  else // otherwise, skip loop code altogether
    while(tok!=NEXT) get_token();
}
// Execute a NEXT statement.
void next()
{
  struct for_stack stckvar;

  stckvar = fpop(); // read the loop info

  variables[stckvar.var]++; // increment control var

  // if done, return
  if(variables[stckvar.var] > stckvar.target) return;
```

```
  fpush(stckvar);  // otherwise, restore the info
  prog = stckvar.loc;  // loop
}

// Push the FOR stack.
void fpush(struct for_stack stckvar)
{
  if(ftos==FOR_NEST)
    serror(TOO_MNY_FOR);
  fstack[ftos] = stckvar;
  ftos++;
}

// Pop the FOR stack.
struct for_stack fpop()
{
  if(ftos==0)
    serror(NEXT_WO_FOR);

  ftos—;
  return(fstack[ftos]);
}
// Execute a simple form of the BASIC INPUT command.
void input()
{
  char var;
  int i;

  get_token(); // see if prompt string is present
  if(token_type==QUOTE) {
    cout << token; // if so, print it and check for comma
    get_token();
    if(*token != ',') serror(SYNTAX);
    get_token();
  }
  else cout << "? "; // otherwise, prompt with ?
  var = toupper(*token)-'A'; // get the input var

  cin >> i; // read input

  variables[var] = i; // store it
}

// Execute a GOSUB command.
void gosub()
{
  char *loc;
```

```
  get_token();

  // find the label to call
  loc = find_label(token);
  if(loc==NULL)
    serror(UNDEF_LAB); // label not defined
  else {
    gpush(prog); // save place to return to
    prog = loc;  // start program running at that loc
  }
}
// Return from GOSUB.
void greturn()
{
   prog = gpop();
}

// Push GOSUB stack.
void gpush(char *s)
{
  if(gtos==SUB_NEST)
    serror(TOO_MNY_GOSUB);

  gstack[gtos] = s;
  gtos++;
}

// Pop GOSUB stack.
char *gpop()
{
  if(gtos==0)
    serror(RET_WO_GOSUB);

  gtos—;
  return(gstack[gtos]);
}
```

Использование Small BASIC

В этом разделе мы рассмотрим несколько программ, которые может выполнить Small BASIC. Обратите внимание на поддержку символов как верхнего, так и нижнего регистров. Вам, конечно, захочется самостоятельно написать несколько программ для Small BASIC. Попробуйте намеренно допускать синтаксические ошибки и понаблюдайте, как Small BASIC будет на них реагировать.

Нижеприведенная программа выполняет все команды, поддерживаемые Small BASIC. Назовите ее TEST.BAS

```
PRINT "This program demonstrates all commands."
FOR X = 1 TO 100
PRINT X; X/2, X; X*X
NEXT
GOSUB 300
PRINT "hello"
INPUT H
IF H<11 THEN GOTO 200
PRINT 12-4/2
PRINT 100
200 A = 100/2
IF A>10 THEN PRINT "this is ok"
PRINT A
PRINT A+34
INPUT H
PRINT H
INPUT "this is a test ",y
PRINT H+Y
END
300 PRINT "this is a subroutine"
    RETURN
```

Если вы назвали интерпретатор языка Small BASIC именем SBASIC, то для запуска этой программы вам потребуется дать следующую команду:

SBASIC TEST.BAS

Small BASIC автоматически загрузит программу и начнет ее выполнение. Следующий образец демонстрирует вложенные подпрограммы:

```
"This program demonstrates nested GOSUBs."
INPUT "enter a number: ", I
GOSUB 100
END
100 FOR T = 1 TO I
  X = X + I
  GOSUB 150
NEXT
RETURN

150 PRINT X;
    RETURN
```

Нижеприведенная программа демонстрирует команду INPUT:

```
print "This program computes the volume of a box."
input "Enter length of first side ", l
input "Enter length of second side ", w
input "Enter length of third side ", d
t = l * w * d
print "Volume is ", t
```

Наконец, последний пример иллюстрирует вложенные циклы FOR:

```
PRINT "This program demonstrates nested FOR loops."
FOR X = 1 TO 100
  FOR Y = 1 TO 10
    PRINT X; Y; X*Y
  NEXT
NEXT
```

Дополнение интерпретатора и расширение его возможностей

Добавлять новые команды к интерпретатору Small BASIC достаточно несложно. Для этого вам просто нужно соблюдать общие требования к формату, продемонстрированные на примере команд, которые рассматривались в этой главе. Для того, чтобы добавить новые типы данных, потребуется использовать массив структур для хранения переменных, причем один из членов структуры должен указывать тип переменной, а второй — хранить ее значение. Для добавления строк вам потребуется создать таблицу строк. Попробуйте реализовать поддержку строк в Small BASIC на базе нового класса **string**. Это позволит выполнять однозначный перевод операций над строками в BASIC в аналогичные операции C++.

Еще одно замечание. В представленном здесь коде интерпретатора и модуля разбора выражений многие перечисления и константы дублируют друг друга в обоих этих файлах. Это удобно для представления в книге, так как в этом случае появляются дополнительные гарантии того, что читатель по невниманию что-то упустит из виду. Однако, по мере того, как вы будете дополнять и расширять свой интерпретатор, вы, возможно, захотите оптимизировать его, и с этой целью создать заголовочный файл, где будут содержаться все эти константы и декларации. Этот заголовочный файл необходимо будет включать во все файлы, входящие в состав вашего интерпретатора. Это не только упростит ваши задачи, возникающие по мере разрастания проекта, но и обеспечит синхронизацию версий файлов.

Разработка собственного языка программирования

Хотя расширение возможностей интерпретатора Small BASIC — хороший способ ознакомиться с его работой и с принципами функционирования языковых интерпретаторов вообще, вы вовсе не ограничены только языком BASIC. Методы, обсуждавшиеся в этой главе, можно применить для написания интерпретатора практически любого языка программирования. Вы можете даже создать собственный язык программирования, который отражал бы ваш стиль мышления и вашу личность. На практике приведенная здесь основа интерпретатора представляет собой хорошую тестовую модель для практически любой новой языковой возможности, которую вам хотелось бы реализовать. Например, для того, чтобы добавить к интерпретатору цикл REPEAT/UNTIL, вам потребуется выполнить следующие действия

1. Добавить REPEAT и UNTIL к перечислению **tokens**
2. Добавить REPEAT и UNTIL в таблицу **commands**
3. Добавить REPEAT и UNTIL в утверждение **switch** в главном цикле программы
4. Определить функции **repeat()** и **until()**, которые будут обрабатывать команды REPEAT и UNTIL (в качестве стартовой точки разработки можно использовать функции **exec_for()** и **next()**.)

Наконец, последнее замечание: типы утверждений, которые вы можете интерпретировать, ограничиваются только вашим воображением. Не бойтесь экспериментировать.

Глава 11

От C++ к Java

В хронике развития языков программирования должно быть сказано примерно следующее: В породил С, С развился в С++, а С++ трансформировался в *Java*. Эта последняя глава посвящена исследованию языка *Java*.)

Каждый, кто так или иначе связан с компьютерным бизнесом, знает, что *World Wide Web* и *Internet* стали неотъемлемой частью компьютерной вселенной. Сеть Internet из обычной системы распределения информации превратилась в глобальную распределенную компьютерную среду. Это и послужило толчком к созданию такого потомка С++, как *Java*. Вот почему это произошло.

Как известно, в сети существуют две широкие категории объектов, которыми обмениваются сервер и ваш ПК, а именно: пассивная информация и активные динамические программы. Например, при чтении электронной почты вы просматриваете пассивные данные. Даже когда вы загружаете программу, ее код будет представлять собой пассивные данные до тех пор, пока вы не начнете ее исполнение. Однако, существует и другой тип объекта, который может быть передан на ваш компьютер — это динамическая программа. Такая программа может быть предоставлена сервером для надлежащего отображения пересылаемых вам данных. Хотя их привлекательность и так велика, сетевые динамические программы вызывают такой интерес еще и в связи с проблемами защищенности и переносимости. Как будет видно из дальнейшего изложения, *Java* представляет собой попытку дать ответ на эти вопросы.

В этой главе дан только краткий обзор языка *Java*. В действительности *Java* представляет собой богатую среду программирования с широкими возможностями. Полное и подробное рассмотрение языка *Java* и его библиотек потребует отдельной большой книги. Цели, преследуемые этой главой, гораздо скромнее — они заключаются в том, чтобы пробудить ваш интерес к этой теме.

Примечание: На момент написания этой книги *Java* все еще является развивающимся языком. Хотя информация, представленная здесь, была точна на момент работы над книгой, нет гарантии, что она не изменится со временем.

Что представляет собой *Java*?

Java - это язык *Internet*. Проект *Java* был основан в 1990 году в *Sun Microsystems* Джеймсом Гослингом (James Gosling), Патриком Ноутоном (Patrick Naughton) и Майком Шериданом (Mike Sheridan). На его разработку потребовалось 5 лет. Значимость *Java* помогает понять следующая аналогия: *Java* представляет собой для *Internet* то же самое, что С++ - для системного программирования. *Java* — это фундамент и основа основ.

Между С++ и *Java* существуют две важные аналогии). Во-первых, *Java* использует синтаксис, аналогичный С++. Так, например, общие формы таких циклов, как **for**, **while** и **loop**, у этих двух языков совпадают. Во-вторых, *Java* обеспечивает поддержку объектно-ориентированного программирования методами, очень похожими на методы С++. Из-за внешнего сходства между *Java* и С++ о *Java* легко можно подумать, как об *Internet*-версии языка С++. Однако, это утверждение не совсем справедливо, так как *Java* имеет и существенные отличия от С++, причем эти отличия фундаментальным образом изменяют характер этого языка. Однако, вам не следует из-за этого беспокоиться. Как программист на С++ вы очень быстро освоитесь с *Java*.

Язык *Java* можно использовать для разработки двух типов программ: приложений (**applications**) и апплетов (**applets**). Приложение представляет собой программу, выполняющуюся на вашем компьютере под управлением операционной системы этого компьютера. Это означает, что приложение, созданное с помощью *Java*, в большей или меньшей степени похоже на приложение, написанное на С++. Апплет же представляет собой приложение, разработанное специально для распространения по *Internet* и должен выполняться на *Java*-совместимом броузере *Web*. Примеры кода, представленные в этой главе, представляют собой самостоятельные приложения. Однако, большинство из изложенных в этой главе методов можно обобщить и на апплеты.

Почему *Java*?

Поскольку С и С++ представляют собой мощные, великолепно разработанные и профессиональные языки, закономерен вопрос: зачем же понадобилось создавать еще один язык программирования? Ответ на этот вопрос умещается в двух словах: защищенность и переносимость. Давайте рассмотрим каждую из этих черт.

Защищенность

Вы наверняка осведомлены о том, что каждый раз, когда вы загружаете "обычную" программу, вы подвергаетесь риску вирусного заражения. До появления *Java* большинство пользователей нечасто загружали исполняемые программы, а те, кто это делал, выполняли проверку на вирусы перед исполнением. Однако, даже в такой ситуации большинство пользователей все же беспокоилось о том, что риск заражения системы вирусом все же не устраняется полностью, и, кроме того, мало кому хотелось допустить свободное проникновение в свою систему программ-злоумышленников. *Java* решает эти проблемы путем создания брандмауэра между сетевым приложением и вашим компьютером.

При использовании *Java*-совместимого броузера **Web** апплеты *Java* можно свободно загружать без риска заражения вирусом. Способ, которым *Java* добивается этого, заключается в том, что апплеты *Java* передаются исполняющей среде *Java* и не получают доступа к остальным частям компьютера. (Очень скоро вы наглядно увидите, как это делается.) Способность загрузки апплетов *Java* с уверенностью в том, что клиентскому компьютеру не будет причинено никакого вреда, представляет собой важнейший аспект языка *Java*.

Переносимость

Во всем мире существует множество разнообразных типов компьютеров и операционных систем — и все они подключены к *Internet*. Для того, чтобы программы могли динамически загружаться на всех типах платформ, соединенных с **Web**, необходимы средства генерации портируемого исполняемого кода. Как вы увидите из дальнейшего, средства, которыми *Java* достигает переносимости, одновременно и эффективны, и элегантны.

Магическое решение Java: Java Bytecode

Ключом, позволяющим *Java* достичь решения проблем переносимости и защищенности, которые мы только что обсудили, является тот факт, что выходной код компилятора *Java* не является исполняемым кодом. Скорее, этот код является байт-кодом. Байт-код представляет собой высокооптимизированный набор инструкций, разработанных для выполнения на виртуальной машине, эмулируемой системой времени выполнения в среде *Java*. Это означает, что система времени выполнения *Java* представляет собой интерпретатор байт-кода. Следовательно, *Java* представляет собой просто набор высокоэффективных средств кодирования программы для интерпретации.

То, что *Java* является интерпретируемым языком, может на первых порах вызвать удивление. Как вам известно, программы, написанные на C++ (и большинстве других современных языков программирования), компилируют

ся в исполняемый объектный код. В действительности, очень немногие языки разрабатываются как интерпретируемые — главным образом из соображений производительности. Однако, тот факт, что программы на *Java* интерпретируются, помогает достичь решения основных проблем, возникающих при загрузке программ через *Internet*. Причины этого мы подробно обсудим.

Поскольку язык *Java* изначально разрабатывался как интерпретируемый, обеспечить возможность работы программ *Java* в большинстве сред стало существенно проще. Причина этого проста: для каждой из платформ необходимо реализовать только систему времени выполнения языка *Java*. Если для конкретной платформы существует система времени выполнения *Java*, байт-код любой программы на *Java* сможет на ней выполняться. Не забывайте, что хотя реализации системы времени выполнения *Java* для разных платформ могут иметь различия, все они интерпретируют один и тот же байт-код *Java*. Если бы программы на *Java* компилировались в исполняемый объектный код, то для каждого из типов процессоров, которые могут иметься в *Internet*, потребовалось бы создавать свою версию одной и той же программы. Это, разумеется, не является приемлемым решением. Таким образом, использование байт-кода для представления программ является простейшим путем к созданию по-настоящему переносимых программ.

То, что *Java* — интерпретируемый язык, помогает достичь не только переносимости, но и защищенности. Поскольку исполнение каждой из программ *Java* находится под контролем системы времени выполнения, система времени выполнения может содержать в себе программу, не позволяя ей создавать побочные эффекты за своими пределами. Кроме того, защищенность усиливается дополнительными ограничениями, существующими в языке *Java*.

Вероятно, вам уже известно, что если программа интерпретируется, то, как правило, она исполняется гораздо медленнее, чем ее откомпилированная версия. Однако, при использовании *Java* разница между этими вариантами не велика. Использование байт-кода позволяет системе времени выполнения *Java* исполнять программы гораздо быстрее, чем можно было бы ожидать.

Наконец, последнее замечание. Хотя *Java* разрабатывался как интерпретируемый язык, нет никаких ограничений, которые не позволяли бы осуществлять "на лету" компиляцию байт-кода в "родной" код системы времени выполнения *Java*. Однако, даже если к байт-коду применить динамическую компиляцию, все характеристики переносимости и защищенности программ на *Java* останутся в силе, поскольку система времени выполнения по-прежнему будет полностью отвечать за выполнение программы.

Различия между *Java* и С++

Хотя язык *Java* разработан на С++, между двумя языками существуют некоторые различия. Некоторые из этих различий представляют собой просто несущественные вариации. Однако, наряду с такими незначительными изменениями

существуют и другие различия, являющиеся следствием ключевых решений разработчиков и оказывающие фундаментальное влияние на способ написания программ. Мы не будем перечислять здесь все различия между C++ и *Java*, однако некоторые из них обсудим довольно подробно. Прежде, чем приступать к дальнейшему обсуждению, еще раз напомним вам тот факт, что язык *Java* разрабатывался с тем, чтобы обеспечить безопасную загрузку переносимых приложений в сетевой среде. Именно это, а не замена С или C++ как языков системного программирования, и было целью создания *Java*. Теперь, запомнив эти основные факты, приступим к рассмотрению некоторых различий между C++ и *Java*.

Какие возможности отсутствуют в Java?

Хотя язык *Java* во многом подобен C++, некоторые из возможностей C++ он не поддерживает. В ряде случаев конкретные возможности C++ просто не связаны со средой *Java*. В других случаях разработчики *Java* намеренно исключили из *Java* дублирование возможностей, существующее в C++. Существуют и такие случаи, когда реализация в *Java* тех или иных возможностей C++ казалась слишком опасной для апплетов *Internet*. Наиболее важные из этих "сокращений" перечислены далее.

Возможно, наиболее существенным отличием *Java* от C++ является то, что *Java* не поддерживает указателей! Как программист на C++ вы хорошо знаете, что указатели являются важнейшим и одним из наиболее мощных средств C++. Однако, при неправильном использовании они становятся и одной из наиболее опасных возможностей языка. Указатели были исключены из *Java* по двум причинам. Во-первых, они не обеспечивают защищенности. Например, использование указателей в стиле C++ позволяет получить доступ к адресам памяти, находящимся за пределами областей кода и данных конкретной программы. Программа-злоумышленник может воспользоваться этим фактом с целью повреждения системы, получения несанкционированного доступа к ее ресурсам (типа подглядывания паролей) или иного нарушения ограничений, наложенных системой безопасности. Во-вторых, даже если указатели ограничить в соответствии с требованиями системы времени выполнения *Java* (что теоретически возможно, так как программы Java интерпретируются), разработчики *Java* сочли указатели потенциальным источником неприятностей. Так или иначе, но указателей в *Java* не существует, и, следовательно, не существует и оператора взятия ссылки (->).

Ниже приведен список наиболее существенных возможностей C++, не реализованных в *Java*.

- *Java* не включает в свой состав структур (structures) и объединений (unions). Разработчики *Java* сочли их избыточными, так как существует более общее понятие класса, которое включает в себя все остальные формы.
- *Java* не поддерживает перегрузки операторов. Разработчики просто не сочли эту возможность достаточно важной.

- *Java* не включает в свой состав препроцессора и не поддерживает его директив. Как известно, в C++ роль препроцессора гораздо менее важна по сравнению с его значимостью для C. Разработчики *Java* сочли, что настало время отказаться от препроцессора полностью.
- *Java* не выполняет никаких автоматических преобразований типа, приводящих к потере точности. Например, чтобы преобразование из *long int* в *int* имело место, его необходимо задать явным образом.
- Весь код программы на *Java* инкапсулирован в один или несколько классов. Поэтому *Java* не содержит ничего похожего на глобальные переменные или глобальные функции.
- *Java* не поддерживает множественного наследования
- Хотя *Java* поддерживает конструкторы, но, наряду с этим, деструкторы не поддерживаются. Вместо деструкторов в *Java* введена функция **finalize( )**.
- *Java* не поддерживает **typedef**.
- В *Java* невозможно декларировать беззнаковые целые (**unsigned int**).
- *Java* не допускает использования оператора **goto**.
- *Java* не содержит оператора **delete**.
- Операторы >> и << не перегружаются для операций ввода/вывода.
- *Java* не поддерживает шаблоны (**templates**).

Что было добавлено в *Java*?

Помимо различий, заключающихся в том, что *Java* не поддерживает некоторых возможностей, которые имеются в C++, есть и другие различия между этими языками. Так, по сравнению с C++, в *Java* был добавлен целых ряд новых возможностей. Возможно, наиболее важной из них является многопоточное программирование (*multithreaded programming*). Как известно, многопоточность позволяет одной или нескольким частям одной и той же программы выполняться на конкурентной основе. Другой важной новой возможностью является подход *Java* к выделению памяти. Как и в C++, в *Java* поддерживается ключевое слово **new**. Однако, ключевое слово **delete** *Java* не поддерживает. Вместо этого после удаления последней ссылки на объект в *Java* удаляется и сам этот объект. Кроме того, *Java* обеспечивает автоматический "сбор мусора" (*garbage collection*), благодаря чему программист не должен делать это вручную. Ниже приведен список наиболее важных новых возможностей *Java*:

- Утверждения **break** и **continue** расширены новыми возможностями и могут воспринимать метки
- Тип **char** теперь декларирует 16-битные символы *Unicode*. Это уподобляет его типу **wchar_t**, имеющемуся в C++. Использование символов *Unicode* упрощает задачу обеспечения переносимости.

- В *Java* добавлен новый оператор >>>, осуществляющий беззнаковый правый сдвиг.
- В дополнение к однострочным и многострочным комментариям, в *Java* добавлен третий тип комментариев — документирующий (*documentation comment*). Документирующие комментарии начинаются с последовательности /** и завершаются последовательностью */.
- *Java* содержит встроенный тип **String**. Этот тип в чем-то похож на стандартный класс **string**, имеющийся в С++. Разумеется, в С++ класс **string** будет доступен только в том случае, если вы включите в свою программу декларацию этого класса. Встроенным типом он не является.

Пример программы на *Java*

Прежде, чем обсуждать дальнейшие теоретические вопросы, рассмотрим рабочий пример программы на языке *Java*. Приведенный здесь пример декларирует 3 переменных, присваивает им значения, после чего отображает их.

```
/* Это - пример простой программы на Java.
   Обратите внимание на поддержку многострочных
   комментариев.
*/

// Java поддерживает и однострочные комментарии

class JavaTest {
  public static void main(String strargs[])
  {
    // Java декларирует встроенные переменные в точности как C++
    int x;
    double y;
    char ch;

    // Присваивания также выполняются в точности как в C++
    x = 10;
    y = 10.23;
    ch = 'X';

    // Однако, вывод на консоль осуществляется с помощью функций
    System.out.print("Это x: ");
    System.out.print(x);
    System.out.println();
    System.out.println("Это y: " + y);
    System.out.println("Это ch: " + ch);
  }
}
```

```
Ниже приведен образец вывода этой программы:
```

Это x: 10

Это y: 10.23

Это ch: X

Давайте внимательно рассмотрим эту программу. На первый взгляд она кажется самой обычной программой на C++. Однако, даже в такой простой программе при более близком рассмотрении можно выявить различия. Во-первых, обратите внимание на то, что функция **main()** декларируется внутри класса **JavaTest**. Как уже упоминалось, весь код *Java* должен располагаться внутри определения класса — и даже функция **main( )**. Обратите внимание на то, что **main( )** декларируется как **public** и **static**. Делается это потому, что, как и в C++, **main( )** является первой функцией, вызываемой, когда программа на *Java* начинает выполняться. Поскольку функция вызывается извне своего класса, поэтому ее необходимо декларировать как **public**. Так как функция вызывается раньше, чем создается какой бы то ни было экземпляр ее класса, ее необходимо декларировать как **static**. В этой ситуации ключевое слово **static** имеет в *Java* тот же смысл, что и в C++: оно позволяет декларировать членов класса, которые существуют до того, как будут созданы какие-либо объекты этого класса. Это позволяет вызывать функцию **main( )** раньше, чем будут созданы какие-нибудь объекты класса **JavaTest**.

Обратите внимание, что параметр функции **main( )** имеет тип **String**. Когда программа начинает выполняться, этот параметр принимает все ассоциированные с программой аргументы командной строки. Первый аргумент будет **String[0]**, второй — **String[1]** и т. д. Этот способ разбора командной строки отличается от C++, где первым аргументом командной строки является имя программы.

В соответствии с комментариями нашего простого примера декларирование переменных и присвоение им значение происходят точно так же, как в C++. Как уже упоминалось ранее, *Java* использует синтаксис C++, поэтому большинство базовых синтаксических конструкций будут выглядеть в *Java* точно так же, как и в C++. Однако, обратите внимание на то, что содержимое переменных выводится с помощью функций **print( )** и **println( )**, а стандартные операторы ввода/вывода, принятые в C++, не используются. В *Java* ввод/вывод на консоль осуществляется путем вызова функций ввода/вывода. Функция **print( )** выводит на экран свой аргумент. Функция **println( )** делает то же самое, но еще и добавляет новую строку. Как вы могли бы ожидать, существуют различные перегруженные формы этих функций, позволяющие выводить любой встроенный тип. Обратите особое внимание на следующую строку программы:

```
System.out.println("Это y: " + y);
```

Обратите внимание на то, что эта команда выводит на экран строку "Это **y:** " и значение переменной **y**. Как можно предположить, в *Java* оператор **+** определен для операций со строками. В данном случае в результате его применения значение y будет получено, автоматически преобразовано в его строковой эквивалент, а затем полученная строка будет конкатенирована с предыдущей строкой.

Методы вместо функций

Терминология языка *Java* редко использует слово "функция". Вместо этого слова программисты на *Java* часто называют "методами" то, к чему программисты на С++ привыкли как к "функциям-членам". Причина этого различия в терминологии заключается в том, чтобы подчеркнуть тот факт, что *Java* не поддерживает глобальных функций. Все функции в *Java* являются членами класса. Поскольку эта книга предназначена как раз для программистов на С++, мы продолжим называть функции именно функциями. Однако, вам следует знать, что в терминологии языка *Java* термин "метод" является предпочтительным.

Компиляция программы на *Java*

Теперь, просмотрев и изучив простую программу на *Java*, вы должны научиться компилировать и исполнять ее. Эти операции не настолько тривиальны, как можно было бы подумать. Во-первых, все программы на *Java* используют расширение имени файла **.java**. Во-вторых, файл, в котором содержится программа, должен иметь точно такое же имя, как и ее класс. Более того, соотношение строчных и прописных букв в имени программы должно быть точно таким же, как и в имени класса. Так, в предыдущем примере имя файла, в котором содержится эта программа, должно быть следующим:

```
JavaTest.java
```

После ввода программы и сохранения ее под надлежащим именем файла, необходимо воспользоваться компилятором *Java*, который называется **javac**, и откомпилировать программу в байт-код. Файл, содержащий байт-код, получит следующее расширение имени файла: **.class**. Для интерпретации этого байт-кода используется интерпретатор *Java*, который носит название **Java**. (Не забывайте о соотношении строчных и прописных букв в именах файлов!) Например, для того, чтобы откомпилировать и выполнить программу, приведенную в предыдущем примере, необходимо выполнить следующую последовательность команд:

```
javac JavaTest.java
java JavaTest
```

Когда программа запускается на выполнение, *Java* ищет указанный класс (в данном случае - **JavaTest**) и интерпретирует его, начиная с функции **main( )**.

Второй пример

Прежде, чем переходить к более продвинутым возможностям *Java*, давайте рассмотрим еще одну простую программу. Нижеприведенный пример демонстрирует расширенные возможности **break** в языке *Java* и выполняет простой ввод с консоли.

```
/* Here is an example that illustrates the expanded
   capabilities of break and performs simple input.
*/

class JavaTest {
  public static void main(String strargs[])
        throws java.io.IOException
  {
    int i, j, k;

    System.out.println("Display ASCII codes.");

lab1: for(;;) {
      System.out.println("Enter a character: ");
      do {
        k = System.in.read();
      } while((char) k == '\n');
      i = k;
      j = 1;
lab2:
      while(i>0) {
        while(i>0) {
          j++;
          if((j%20)==0) break;
          if((char) k == 'q') break lab1;
          if((char) k == '\n') break lab2;
          System.out.print(i+" ");
          i—;
        }
        System.out.println();
        j = 1;
      }
    }
  }
}
```

Эта программа предлагает пользователю ввести символ. Затем она отображает все ASCII-коды, начиная с этого символа в нисходящем порядке до нуля. Пример вывода приведен ниже:

```
Display ASCII codes.
Enter a character:
1
49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14
13 12 11 10 9 8 7 6 5 4 3 2 1
```

```
Enter a character:
2
50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33
32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15
14 13 12 11 10 9 8 7 6 5 4 3 2 1
Enter a character:
3
51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34
33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Enter a character:
q
```

В языке *Java* команда **break** может воспринимать метку как аргумент. Если метки в качестве аргумента не указано, то действие **break** будет аналогично действию команды **break** в C++. Однако, если метка присутствует, то управление передается из блока с указанной меткой. В вышеприведенном примере оператор **break** сам по себе, без метки, просто осуществляет выход из внутреннего цикла **while**. Оператор **break lab2** осуществляет выход из внешнего цикла **while**. Оператор **break lab1** осуществляет выход из внешнего цикла **for**, вызывая завершение программы при вводе символа *q*. Как можно было предположить, способность **break** воспринимать метку как аргумент существенно расширяет диапазон применения этого оператора. Фактически, одной из причин исключения оператора **goto** из *Java* послужили расширенные возможности оператора **break**.

Способность **break** воспринимать метки можно применить и к выходу из утверждения **switch**. Утверждение **continue** также может продолжаться помеченным блоком. Запомните, что метка, которая служит аргументом для утверждений **break** или **contunue**, должна находиться в начале блока.

Кроме того, по этому примеру можно сделать еще несколько важных замечаний. Во-первых, обратите внимание на то, что символы считываются путем вызова функции **System.in.read( )**. Эта функция считывает символы из стандартного вводного потока. По умолчанию стандартный вводной поток линейно буферизуется, поэтому вы должны нажать клавишу ENTER прежде, чем какие-либо символы, введенные вами, будут направлены вашей программе. Эта ситуация аналогична существующей в C++, и вы, возможно, уже знакомы с этим стилем ввода. Фактически, большинство реальных приложений *Java* будут графическими, поэтому эта черта обеспечивает только минимальную поддержку консольного ввода/вывода. Еще одно замечание: поскольку используется функция **System.in.read( ),** в программе необходимо указать исключение **throw java.io.IOException**.

Работа с классами

Что касается особенностей работы с классами, то *Java* в этом отношении разделяет многие особенности C++, однако, и здесь тоже существуют различия. Хотя обсуждение многих тонкостей и нюансов далеко выходит за рамки данной книги, мы опишем здесь наиболее важные аспекты работы с классами в *Java*.

Начнем с установления сходства между классами С++ и классами *Java*. Классы *Java* могут содержать как переменные-члены, так и функции-члены (которые в *Java* предпочтительнее называть методами). Классы *Java* могут включать конструкторы. Конструкторы могут перегружаться. Фактически, перегружаться может любая функция-член. Перегрузка в *Java* работает более или менее аналогично перегрузке в С++. Каждый создаваемый объект будет иметь собственную копию своих переменных-членов (еще одна черта, сходная с С++).

Теперь перейдем к различиям. Начнем с изучения следующей программы:

```
// Creating Class Objects
class JavaTest {
  private int a;
  JavaTest() { a = 0; } // constructors
  JavaTest(int i) { a = i; }

  int geta() { return a; } // a simple method

  public static void main(String strargs[])
  {
    // instantiate JavaTest objects
    JavaTest t1 = new JavaTest();
    JavaTest t2 = new JavaTest(100);

    System.out.println("This is t1's a: " + t1.a);
    System.out.println("This is t2's a: " + t2.a);
  }
}
```

Во-первых, обратите внимание на то, что переменной-члену **a** предшествует определитель доступа **private**. По умолчанию к членам класса могут получить доступ только другие члены этого класса, производные классы и другие члены их пакета (**package**). (Пакеты будут обсуждаться далее.) Таким образом, по умолчанию, к членам класса в *Java* в большей степени применима характеристика "**public**", чем в С++, однако, свойствами **public** они обладают не в полной мере. Определяя **a** как **private**, мы ограничиваем эту переменную диапазоном ее класса (как и в С++). Однако в отличие от С++ определитель доступа **private** применяется только к переменной или методу, которому он непосредственно предшествует. То же самое справедливо и для остальных определителей доступа. *Java* не использует конструкций типа **private:** (и так далее). В этой программе **a** декларируется как **private** только в иллюстративных целях.

Внутри функции **main( )** обратите особое внимание на то, как создаются объекты **JavaTest**. Рассмотрим, например, следующую строку:

```
JavaTest t1 = new JavaTest();
```

Она создает новый объект типа **JavaTest** и инициализирует его, используя без параметров конструктор **JavaTest( ).** Переменная *t1* представляет собой ссылку на этот объект. Очень важно понимать, что в *Java* нижеприведенное утверждение не создает никакого объекта:

```
JavaTest ob;
```

Это утверждение создает только ссылку на объект **JavaTest**. Для того, чтобы создать сам объект, вы должны выделить под него пространство, используя утверждение **new**. Например, нижеприведенное утверждение связывает объект со ссылкой **ob**:

```
ob = JavaTest(99);
```

Не забывайте, что экземпляры всех объектов классов в *Java* создаются с помощью оператора **new**. Таким образом, память для всех объектов класса выделяется динамически. В *Java* осуществляется автоматический "сбор мусора", и если на какой-либо объект нет ссылок, он будет считаться неактивным и будет удален при следующей операции "сбора мусора", чтобы таким образом освободить память.

Финализаторы

Хотя в *Java* имеются конструкторы, такого понятия, как деструкторы, там не существует. Вместо деструкторов вы можете декларировать финализатор (*finalizer*). Финализатор вызывается в том случае, когда объект класса должен быть удален в ходе очередной операции по "сбору мусора". Как видите, концепция финализатора подобна концепции деструктора, однако при исполнении они фундаментально различаются. Далее, если объект никогда не удаляется в процессе "сбора мусора", финализатор никогда не вызывается. В то же время в C++ деструктор вызывается, когда объект выходит из диапазона. В *Java* финализатор вызывается только в том случае, когда объект должен быть удален в процессе "сбора мусора".

Финализаторы имеют следующую общую форму:

```
protected void finalize( )
{
    // Здесь находится код финализатора
}
```

Ниже приведен предыдущий пример со схематическим представлением финализатора:

```
// Creating Class Objects
class JavaTest {
  private int a;
```

```
  JavaTest() { a = 0; } // constructors
  JavaTest(int i) { a = i; }

  int geta() { return a; } // a simple method

  /* This is called when an object is about to be
     subject to garbage collection. */
  protected void finalize()
  {
    // shutdown code here
  }

  public static void main(String strargs[])
  {
    JavaTest t1; t1 = new JavaTest();
    JavaTest t2 = new JavaTest(100);

    System.out.println("This is t1's a: " + t1.a);
    System.out.println("This is t2's a: " + t2.a);
  }
}
```

Иерархия классов *Java*

Подобно С++, *Java* поддерживает иерархию классов. Фактически эта иерархия является критическим компонентом *Java*. Однако, способ ее реализации в *Java* существенно отличается от способа, принятого в С++. Во-первых, как уже упоминалось, множественное наследование в *Java* не поддерживается. Это означает, что все иерархии классов *Java* будут линейными. Во-вторых, *Java* использует ключевое слово **extends** для указания базового класса. Рассмотрим следующий пример

```
// Inheritance, Java style.

// Here is a base class.
class MyClass {
  int a;
  int geta() { return a; }
  int add(MyClass ob)
  {
     return a + ob.a;
  }
}
```

```
// JavaTest inherits MyClass
class JavaTest extends MyClass {
  int b;
  public static void main(String strargs[])
  {
    JavaTest t1 = new JavaTest();
    JavaTest t2 = new JavaTest();
    JavaTest t3 = new JavaTest();

    t1.a = 0;
    t1.b = 100;

    t2.a = 20;
    t2.b = 200;

    t3.a = 30;
    t3.b = 300;

    System.out.print("This is t1's a and b: ");
    System.out.println(t1.a + " " + t1.b);
    System.out.print("This is t2's a and b: ");
    System.out.println(t2.a + " " + t2.b);
    System.out.print("This is t3's a and b: ");
    System.out.println(t3.a + " " + t3.b);

    // call another member function
    int sum;
    sum = t3.add(t2);
    System.out.println("t2+t3 is " + sum);
  }
}
```

Обратите внимание на использование ключевого слова **extends** для наследования базовому классу. Как уже упоминалось, по умолчанию члены базового класса доступны в пределах производного класса, что и иллюстрирует эта программа.

Если базовый класс содержит один или несколько конструкторов, то производный класс должен обеспечить механизм, в соответствии с которым вызываются конструкторы базового класса. Это делается с помощью ключевого слова **super**, использование которого иллюстрирует нижеприведенная расширенная версия предыдущего примера:

```
// Inheritance, Java style.

// Here is a base class.
class MyClass {
  int a;
```

```
  // MyClass Constructors
  MyClass() { a = 0; }
  MyClass(int i) { a = i; }

  int geta() { return a; }

  int add(MyClass ob)
  {
     return a + ob.a;
  }
}

// JavaTest inherits MyClass
class JavaTest extends MyClass {
  int b;

  // these call MyClass Constructors
  JavaTest(int i) { super(i); }
  JavaTest() { super(0); }

  public static void main(String strargs[])
  {
    JavaTest t1 = new JavaTest();
    JavaTest t2 = new JavaTest(20);
    JavaTest t3 = new JavaTest(30);

    t1.b = 100;
    t2.b = 200;
    t3.b = 300;

    System.out.print("This is t1's a and b: ");
    System.out.println(t1.a + " " + t1.b);
    System.out.print("This is t2's a and b: ");
    System.out.println(t2.a + " " + t2.b);
    System.out.print("This is t3's a and b: ");
    System.out.println(t3.a + " " + t3.b);

    // call another member function
    int sum;
    sum = t3.add(t2);
    System.out.println("t2+t3 is " + sum);
  }
}
```

Поскольку *Java* не поддерживает множественного наследования, ключевое слово **super** всегда относится к непосредственно предшествующему базовому классу Как видите, оно вызывает конструкторы базового класса, передавая аргументы по мерс надобности.

Наконец, последнее замечание о наследовании. В терминологии *Java* о нем говорят как о создании подклассов (**sublassing**). Далее, то, что программисты на C++ обычно называют базовым классом, программисты *Java* называют суперклассом (**superclass**). То, что программисты на C++ называют производным классом, в *Java* называется подклассом (**subclass**).

Классы и файлы

Как правило, противопоказаний к тому, чтобы держать базовый класс и его производные классы в одном файле *Java* не существует. Однако, возможны такие ситуации, когда программисту этого не хочется. В качестве примера можно привести такой случай, когда базовый класс должен расширяться подклассами в различных файлах. В этом случае будет рационально поместить базовый класс в отдельный файл и назвать его тем же именем, что и базовый класс. После этого базовый класс будет автоматически извлекаться из своего файла при каждом обращении к нему. Попробуйте реализовать следующий пример. Разбейте предыдущую программу на два файла. Поместите **MyClass** в файл **MyClass.java**. Класс **JavaTest** поместите в файл **JavaTest.java**. Теперь откомпилируйте сначала **MyClass.java**, а затем - **JavaTest.java**. Вы увидите, что при компиляции **JavaTest.java** компилятор автоматически включит и **MyClass**.

Пакеты и импорт

Java содержит два средства управления классами, которые помогают использовать классы и облегчают задачу их организации. Первое из них называется пакетом (**package**). Пакет определяет диапазон. Таким образом, имена, декларированные в пределах пакета, являются для этого пакета защищенными (**private**), если только они не были явным образом объявлены как **public**. Пакет декларируется с помощью ключевого слова **package**. Для хранения пакетов *Java* использует каталоги. По этой причине, каждый пакет должен храниться в каталоге, который имеет то же имя, что и пакет (включая различия между строчными и прописными буквами). Например, в нижеприведенном примере **MyClass** модифицирован для использования в пакете, который называется **MyPack**:

```
package MyPack; // this declares a package
public class MyClass {
  public int a;
  public MyClass() { a = 0; } // constructors
  public MyClass(int i) { a = i; }

  public int geta() { return a; }
  public int add(MyClass ob)
```

```
  {
     return a + ob.a;
  }
}
```

Не забывайте, что теперь этот файл должен располагаться в каталоге с именем **MyPack**. Члены класса **MyClass** объявлены как **public**, чтобы обеспечить их доступность за пределами ограничений **MyPack**. В реальной программе большинство членов пакета останется **private**.

При использовании вышеприведенного пакета, первая строка **JavaTest** должна теперь выглядеть следующим образом:

```
class JavaTest extends MyPack.MyClass {
```

Как видите, теперь **MyPack** предшествует классу **MyClass**.

Вы можете не указывать каждый класс по отдельности, а вместо этого импортировать целый пакет. Это делается с помощью ключевого слова **import**. Например, если вы хотите импортировать класс **MyClass** из пакета **MyPackage**, необходимо воспользоваться следующим утверждением:

```
import MyPack.MyClass;
```

После этого вы сможете расширить **MyClass** непосредственно, и вам не понадобится указывать **MyPack** при создании **JavaTest**.

Если пакет включает несколько классов (ситуация, наиболее частая при реальном программировании), вы можете включить все эти классы, указав вместо имени класса символ *:

```
import MyOtherPack.*; // import all classes
```

Это утверждение импортирует все классы **public**, содержащиеся в пакете **MyOtherPack**.

Интерфейсы

Ключевое слово **interface** позволяет определить внешнюю форму класса, не определяя его поведение. Определение интерфейса в чем-то напоминает декларирование абстрактного класса в С++. В интерфейсе не декларируется никаких переменных-членов, и объявляются только прототипы функций-членов. (Таким образом, декларации функций в интерфейсе в чем-то подобны чистым виртуальным функциям С++.) После того, как интерфейс разработан, класс может его реализовать. Для этого в классе должно быть создано несколько методов, реализующих все части интерфейса. Пример такой реализации мы приводить не бу

дем, так как это выходит за рамки материала данной книги. Здесь достаточно будет сказать, что интерфейсы в *Java* служат (более или менее) той же цели, что и виртуальные функции и абстрактные классы в C++.

Стандартные классы

Java содержит несколько стандартных пакетов классов, которые обеспечивают довольно широкие встроенные возможности в области ввода/вывода, графики и т. п. В нижеприведенной таблице вы найдете некоторые из стандартных пакетов *Java*.

Пакет	Назначение
java.applet	Поддержка апплетов
java.awt	Набор средств для работы с абстрактными окнами
java.lang	Языковая поддержка
java.io	Поддержка ввода/вывода
java.net	Сетевая поддержка
java.util	Стандартные классы типа Stack, Vector и Date

Набор средств для работы с абстрактными окнами используется для разработки приложений, работающих в оконной среде, такой, как Windows 95 или Windows NT.

Рекомендации для самостоятельной разработки

Если вы заинтересованы в написании программ для *Internet*, вам потребуются фундаментальные знания в области *Java*. Поскольку этот язык предлагает надежный и безопасный путь передачи приложений по *Internet*, он почти наверняка станет тем языком, с которым вы будете работать. Поэтому автор считает нужным порекомендовать для прочтения следующую книгу: *The Java Handbook by Patrick Naughton* (Osborne/McGraw-Hill, Berkeley, Ca 1996). Патрик работает с *Java* с самого начала этого проекта, и его опыт будет просто бесценен для всех, кто изучает этот язык.

Первое, что вам захочется сделать, ознакомившись с программированием на *Java*, это разработка собственных апплетов и отправка их по **Web**. Помните, апплеты являются движущей силой *Java*.

Как уже говорилось в начале этой главы, *Java* продолжает быстро развиваться. Если навигация по Сети доставляет вам наслаждение, поищите там информацию о последних новшествах в связи с проектом *Java*.

Приложение A

Сводка ключевых слов C++

Это приложение содержит краткое описание 62 ключевых слов C++. Все эти ключевые слова приведены в таблице A-1. Все ключевые слова C++ используют строчные буквы. C++ различает строчные и прописные буквы, поэтому, например, **else** является ключевым словом, а **ELSE** — нет. Далее приведен алфавитный список всех ключевых слов.

Таблица A-1. Ключевые слова C++

asm	auto	bool	break
case	cath	char	class
const	const_cast	continue	default
delete	do	double	dynamic_cast
else	enum	explicit	extern
false	float	for	friend
goto	if	inline	int
long	mutable	namespace	new
operator	private	protected	public
register	reinterpret_cast	return	short
signed	sizeof	static	static_cast
struct	switch	template	this
throw	true	try	typedef
typeid	tapename	union	unsigned
using	virtual	void	volatile
wchar_t	while		

asm

Ключевое слово **asm** используется для встраивания команд на языке ассемблера в программу на C++. Общий формат выглядит следующим образом:

asm *op-code*;

где *op-code* - встраиваемая инструкция.

auto

auto используется для создания локальных переменных. Использование auto оставляется полностью на усмотрение программиста, так как все локальные переменные имеют этот тип по умолчанию. Поэтому **auto** используется крайне редко.

bool

bool указывает на булевский (или логический) тип. Переменные типа **bool** могут принимать только два значения — истина (**true**) и ложь (**false**).

break

break используется для выхода из циклов **do, for и while**, в обход нормальных условий цикла. Кроме того, это ключевое слово используется также для выхода из утверждения **switch**. Ниже приведен пример выхода из цикла с помощью ключевого слова break:

```
while(x<100) {
  x = get_new_x();
  if(cancel()) break; // exit if canceled
  process(x);
}
```

Здесь, если функция **cancel()** возвращает true, цикл будет завершен вне зависимости от того, какое значение имеет переменная x.

break всегда завершает внутренний, самый глубоко вложенные цикл **for, do** или **while,** или утверждение switch, вне зависимости от вложенности. При использовании с утверждением **switch, break** эффективно предотвращает "проваливание" программы к следующему оператору **case**. (Подробную информацию см. в пояснении к ключевому слову **switch**.)

case

См. **switch**.

catch

Утверждение **catch** является частью механизма обработки исключений С++. Более подробную информацию см. в пояснении к ключевому слову **throw**.

char

char указывает на тип 8-битного символа. Например, для того, чтобы переменная **ch** имела символьный тип, необходимо написать:

```
char ch;
```

В С++ символ имеет длину 1 байт.

class

class формирует базис для объектно-ориентированного программирования и представляет собой фундаментальную единицу инкапсуляции в С++. **Class** используется для определения новых типов данных, называемых классами. Синтаксис декларации **class** аналогичен синтаксису декларации структуры. В наиболее общей форме он приведен ниже:

class *class-name* {
  private functions and variables
public:
  public functions and variables
} *object-list;*

В декларации класса параметр *object-list* является опционным. Объекты класса можно декларировать позднее, по мере надобности. Параметр *class-name* технически тоже не является обязательным, но с практической точки зрения он нужен почти всегда. Причина заключается в том, что этот параметр становится именем нового типа данных и используется для последующего декларирования объектов этого класса. Обратите внимание, что декларация класса может включать как переменные, так и функции.

Функции и переменные декларируются внутри декларации класса и называются членами (members) этого класса. По умолчанию, все переменные и функции, объявленные внутри класса, являются по отношению к этому классу закрытыми (private). Это означает, что доступ к ним могут получить только другие члены этого же класса. Для декларации общедоступных (public) членов класса используется ключевое слово **public**, за которым должно следовать двоеточие. Все функции и переменные, которым предшествует спецификатор **public**, доступны как для членов своего класса, так и для любой другой части программы. Кроме того, можно декларировать защищенные (protected) члены класса с помощью ключевого слова **protected**. Доступ к защищенным членам класса могут получать только члены этого же класса и классов, производных от него.

Ниже приведен простой пример декларации класса:

```
class myclass {
  // private to myclass
  int a;
public:
  // public members
  void set_a(int num) { a = num; }
  int get_a() { return a; }
};
```

Доступ к членам класса осуществляется с помощью операторов точка (.) и стрелка (->).

const

Модификатор **const** сообщает компилятору, что переменная, которая за ним следует, не может быть модифицирована. Кроме того, он используется для того, чтобы не позволять функции модифицировать объект, на который дается ссылка одним из ее аргументов. Далее, **const** можно использовать для декларирования функций-членов.

const_cast

const_cast выполняет операцию преобразования типа, имеющую приоритет перед **const** и **volatile**.

continue

continue используется для обхода оставшихся частей кода в цикле, осуществляя непосредственный переход к проверке условия. Например, нижеприведенный цикл не отобразит числа 10:

```
for(x = 0; i<100; x++) {
  if(x==10) continue;
  cout << x << ' ';
}
```

default

default используется в операторе **switch** для указания утверждений, выполняющихся по умолчанию. Подробную информацию можно найти в примечании к оператору **switch**.

delete

Оператор **delete** освобождает память, выделенную с помощью ключевого слова **new**. Подробную информацию можно найти в пояснении к **new**.

do

do представляет собой одну из трех конструкций построения цикла, доступных в С++. Общая форма цикла **do** выглядит следующим образом:

do {
  statement block
} *while(expression);*

Если внутри цикла имеется только одно утверждение, фигурные скобки не являются технически необходимыми, однако, они улучшают читаемость программы. Цикл будет повторяться до тех пор, пока значение выражения истинно.

Цикл **do** является единственным из циклов С++, который всегда будет иметь хотя бы одну итерацию. Это является следствием того, что проверка условия выполняется в конце цикла.

Как правило, цикл **do** используется для чтения файлов с диска. Ниже приведен фрагмент кода, которые будет считывать файл до тех пор, пока не встретит символ конца файла (EOF):

```
do {
  myfile.getc(ch);
  cout << ch;
} while(!myfile.eof());
```

double

double задает тип данных двойной точности с плавающей точкой. Например, для того, чтобы продекларировать переменную **d** как имеющую тип **double**, необходимо написать следующее утверждение:

```
double d;
```

dynamic_cast

dynamic_cast выполняет полиморфичное преобразование типа во время выполнения с подтверждением допустимости преобразования. Если преобразование недопустимо, то операция завершается неудачей, а выражению присваивается значение null. Основным назначением **dynamic_cast** является выполнение преобразований полиморфичных типов. Например, **dynamic_cast** можно использовать для определения совместимости типа объекта, на который указывает указатель, и конечного типа преобразования. При этом, если объект, на который указывает указатель, не принадлежит к конечному типу преобразования или одному из его производных классов, операция **dynamic_cast** завершается неудачей, и результат ее выполнения приравнивается к нулю.

else

См. **if**.

enum

Спецификатор типа **enum** создает тип перечисления. Перечисление представляет собой просто список объектов. Следовательно, тип перечисления определяет то, из чего состоит этот список объектов. Например, нижеприведенный код определяет перечисление с именем **color** и переменную этого типа с именем **c**, после чего выполняет присваивание и реляционную проверку.

```
#include <iostream.h>

enum color {red, green, yellow};
enum color c;

main()
{
  c = red;
  if(c==red) cout << "color is red\n";

  return 0;
}
```

explicit

Ключевое слов **explicit** используется для декларирования неконвертируемых конструкторов. Например, предположим, что у нас есть следующий класс:

```
class MyClass {
  int i;
public:
  MyClass(int j) {i = j;}
  // ...
};
```

то следующее утверждение:

```
MyClass ob2 = 10;
```

будет автоматически преобразовано в форму:

```
MyClass ob2(10);
```

Однако, если объявить конструктор **MyClass** как **explicit**, то автоматическое преобразование не будет иметь места. Ниже приведен текст **MyClass** с использованием конструктора **explicit**:

```
class MyClass {
  int i;
public:
  explicit MyClass(int j) {i = j;}
  // ...
};
```

extern

Модификатор типа данных **extern** используется для того, чтобы сообщить компилятору о том, что переменная декларируется где-то в другой точке программы. Этот метод часто применяется при использовании отдельно компилируемых файлов, которые разделяют одни и те же глобальные данные и связаны вместе на этапе редактирования связей. В частности, он извещает компилятор о наличии переменной без ее повторной декларации. Допустим, если переменная **first** декларировалась в одном из файлов как int, то все последующие файлы будут использовать следующую форму декларации:

```
extern int first;
```

false

false представляет собой одно из двух значений, которые может принимать булевская переменная.

float

float задает тип переменной с плавающей точкой одиночной точности. Например, для того, чтобы продекларировать переменную **f** как имеющую тип **float**, необходимо указать следующее утверждение:

```
float f;
```

for

Цикл **for** позволяет осуществить автоматическую инициализацию и инкрементное приращение переменной-счетчика. Общий формат цикла **for** выглядит следующим образом:

for(***initialization; condition; increment***) {
  statements
}

Если в теле цикла имеется только одно утверждение, то фигурные скобки не обязательны.

Хотя допускаются различные вариации цикла **for**, как правило, инициализация используется для установки переменной-счетчика на начальное значение. Условие цикла, как правило, представляет собой реляционное утверждение, которое

сравнивает значение счетчика со значением, задающим условие выхода из цикла, после чего переменной-счетчику дается положительное или отрицательное приращение.

Например, нижеприведенный код 10 раз выводит на экран слово **hello**:

```
for(t=0; t<10; t++) cout << "hello\n";
```

friend

Спецификатор доступа **friend** предоставляет функции, не являющейся членом класса, доступ к закрытым и защищенным членам класса. Функции-"друзья" не имеют указателей **this** и не применяются в отношении объекта. Они вызываются так же, как и всякие другие нормальные функции.

goto

goto вызывает переход в процессе выполнения программы к метке, указанной в операторе **goto**. Оператор **goto** имеет следующую общую форму:

goto *label;*
.
.
.
label:

Все метки должны заканчиваться двоеточием и не должны конфликтовать с ключевыми словами и именами функций. Далее, **goto** может ветвиться только в пределах текущей функции, а переходы из одной функции в другую не допускаются.

Ниже приведен пример, который распечатывает сообщение **right** и не распечатывает сообщение **wrong**:

```
goto lab1;
  cout << "wrong";
lab1:
  cout << "right";
```

if

if представляет собой оператор условного перехода. Он имеет следующую общую форму:

```
if(expression) {
  statement block 1
}
else {
  statement block 2
}
```

При использовании одиночных утверждений скобки не являются обязательными. Кроме того, само утверждение **else** также не обязательно.

Выражение, управляющее циклом **if,** может принадлежать к любому типу, кроме **void**. Если в результате вычисления этого выражения получается значение, отличное от 0, то будет выполнен блок 1. В противном случае будет выполнен блок 2 (если он существует).

Нижеприведенный фрагмент кода иллюстрирует использование **if**:

```
if(ch < 10) cout << "Less than 10.\n";
else cout << "Greater than or equal to 10.\n";
```

inline

Модификатор **inline** требует, чтобы код функции был встроенным, а не вызываемым. Это снижает количество избыточных операций процессора, связанных с механизмом вызова функций и возврата управления. Однако, встраивание больших функций имеет побочный эффект увеличения объема вашего кода. Функции-члены класса встроены по умолчанию.

int

int определяет тип целых; Например, для того, чтобы декларировать переменную **count** как целую, необходимо ввести следующее:

```
int count;
```

long

long определяет тип длинных целых. Например, для того, чтобы декларировать переменную **count** как длинную целую, необходимо ввести следующее:

```
long int count;
```

mutable

Модификатор **mutable** используется для удаления постоянства с отдельных членов класса при создании копии объекта **const** этого класса.

namespace

Ключевое слово **namespace** определяет диапазон. Общая форма утверждения **namespace** приведена ниже:

namespace *name* {
  // object declarations
}

Например:

```
namespace MyNameSpace {
  int i, k;
  void myfunc(int j) { cout << j; }
}
```

Здесь **i, k,** и **j**, а также **myfunc**() представляют собой часть диапазона, определенного пространством имен **MyNameSpace**.

Поскольку пространство имен определяет диапазон, вы должны использовать оператор разрешения диапазона для ссылок на объекты, определенные в пределах пространства имен. Например, для того, чтобы присвоить переменной **i** значение 10, необходимо использовать следующее утверждение:

```
MyNameSpace::i = 10;
```

Если члены пространства имен будут часто использоваться, вы можете использовать для упрощения доступа директиву **using.** Утверждение **using** имеет две общих формы:

using namespace *name*;
using *name::member;*

В первом случае параметр *name* указывает имя пространства имен, к которому требуется получить доступ. После этого все члены, определенные в пределах этого пространства имен, можно использовать без квалификатора. Во второй форме видимым делается только конкретный член пространства имен. Например, если предположить, что диапазон **MyNameSpace** определен именно так, как было показано в вышеприведенном примере, то справедливы будут следующие утверждения и операции присваивания:

```
using MyNameSpace::k; // only k is made visible
k = 10; // OK because k is visible

using namespace MyNameSpace; // all members are visible
i = 10; // OK because all members of MyNameSpace are now
visible
```

new

Оператор **new** осуществляет выделение памяти. Его общая форма показана ниже:

ptr = new *type;*

Здесь *type* обозначает имя типа данных, для которого производится выделение памяти, а *ptr* должен быть указателем на переменную, тип которой совместим с типом type. Оператор **new** автоматически выделяет объем памяти, достаточный для хранения объекта указанного типа, и возвращает указатель на нее. Если память для хранения объекта выделить невозможно (допустим, из-за ее нехватки), то будет возвращен нулевой указатель. Это означает, что прежде, чем использовать указатель *ptr*, необходимо всегда проверять его значение.

Для того, чтобы освободить ранее выделенную память, используется оператор **delete**. Он имеет следующую общую форму:

delete *ptr;*

Здесь *ptr* представляет собой указатель на память, выделенную путем вызова **new**. Ниже приведен пример, в котором выделяется память для хранения данных типа **float**:

```
float *fptr;

fptr = new float;
if(!fptr) {
  cout << "Allocation error.\n';
  exit(1);
}
// ...
delete fptr;
```

Для выделения памяти под хранение массива используется следующая форма оператора **new:**

ptr = new type[*size*];

где *size* определяет количество элементов типа *type*, имеющихся в массиве. Для освобождения памяти, выделенной под хранение массива, используется следующая форма **delete**:

delete [ ] *ptr*;

operator

Ключевое слово **operator** используется для создания операторов-функций. Функции-операторы, являющиеся членами класса, имеют следующую форму:

ret-type class-name::operator#(...) { *body of function* }

где # представляет собой символ шаблона, заменяемый реальным оператором. Функции-операторы, не являющиеся членами класса, имеют следующую форму:

ret-type operator#(...) { *body of function* }

Для функций-членов, представляющих бинарные операторы, левый операнд передается через **this**, а правый — явным образом через параметр. Для функций-членов, представляющих унарные операторы, операнд передается через **this**.

Для функций бинарных операторов, которые не являются членами класса, левый операнд передается первым параметром, а правый - вторым параметром. Для унарных операторов, не являющихся членами класса, операнд передается как единственный параметр функции.

private

Модификатор доступа **private** используется для декларирования закрытых членов класса. Доступ к таким членам класса могут получить только другие члены этого класса.

protected

Модификатор доступа **protected** используется для декларирования защищенных членов класса. Доступ к таким членам класса могут получить только другие члены этого класса или члены производного класса.

public

Модификатор доступа **public** используется для декларирования общедоступных членов класса. Доступ к таким членам класса можно получить из любой другой части программы.

register

Модификатор **register** требует, чтобы переменная хранилась таким образом, чтобы обеспечить быстрейший возможный доступ. В случае с символами и переменными это обычно означает использование одного из регистров процессора. Для других типов объектов это может означать кэширование памяти. Для того, чтобы декларировать **i** как регистровое целое, необходимо написать:

```
register int i;
```

reiterpret_cast

Оператор **reinterpret_cast** изменит тип на другой, фундаментально от него отличающийся. Например, **reiterpret_cast** можно использовать для преобразования указателя в целое. Таким образом, **reiterpret_cast** может использоваться для взаимного преобразования внутренне несовместимых типов.

return

Утверждение **return** приводит к возврату из функционального вызова и может использоваться для передачи значений вызывающей процедуре. Например, следующая функция возвращает произведение своих целых аргументов:

```
mul(int a, int b)
{
  return a*b;
}
```

Не следует забывать, что как только в коде функции встречается оператор **return**, выполнение функции завершается, а весь остальной код будет пропущен.

short

Модификатор **short** определяет тип целой переменной. Например, для декларации переменной **sh** как короткого целого необходимо указать:

```
short int sh;
```

signed

Модификатор типа **signed** наиболее широко используется для определения знакового символьного типа данных. Его можно использовать для любого из целых типов, но такое определение будет избыточным.

sizeof

Ключевое слово **sizeof** представляет собой оператор времени компиляции. Он возвращает выраженную в байтах длину переменной, которой он предшествует. Если оператор **sizeof** предшествует определению типа, тип должен быть заключен в скобки. Например, в результате выполнения следующего утверждения:

```
cout << sizeof(int);
```

можно получить 2 или 4 в зависимости от среды и компилятора С++.

Принципиальное значение **sizeof** заключается в том, что этот оператор позволяет генерировать переносимый код С++ в тех случаях, когда переносимость кода зависит от размерности встроенных типов данных С++.

static

Модификатор типа данных **static** сообщает компилятору о необходимости создания постоянной области хранения для локальной переменной, которой он предшествует. Это позволяет указанной переменной сохранять свое значение в перерывах между функциональными вызовами. К примеру, для того, чтобы объявить переменную **last_time** как статическое целое, необходимо указать:

```
static int last_time;
```

Модификатор типа **static** может применяться и к членам класса. При создании члена класса, имеющего тип **static**, будет создана только одна копия этого члена, которая будет разделяемой между всеми экземплярами этого класса.

static_cast

Оператор **static_cast** выполняет неполиморфичное преобразование типов. Его можно использовать для любых стандартных преобразований.

struct

Оператор **struct** создает смешанный тип данных, называемый структурой и состоящий из одного или нескольких членов. Ниже приведена простейшая форма структуры:

```
struct struct_name {
  type member1;
  type member2;
  .
  .
  .
  type memberN;
} object-list;
```

Доступ к отдельным членам структуры осуществляется с помощью операций (.) и (->).

По умолчанию, все члены структуры имеют тип public. Однако, для ограничения доступа при декларировании структуры можно использовать модификатор **private** и **protected**. Формально структура создает тип класса.

switch

Утверждение **switch** в С++ представляет собой оператор условного перехода с множественным ветвлением. Общая форма утверждения switch приведена ниже:

```
switch(expression) {
  case constant1:
    statement sequence
    break;
  case constant2:
    statement sequence
    break;
.
.
.
case constantN:
  statement sequence
  break;
 default:
   default statement sequence
}
```

Каждая последовательность утверждений statement sequence может включать в свой состав одно или несколько утверждений. Раздел **default** является необязательным.

Утверждение **switch** работает по принципу сравнения контрольного выражения с константами. Когда контрольное выражение совпадает с одной из констант, следующий за константой набор утверждений будет выполняться до тех пор, пока не встретится оператор **break**. Если оператор **break** пропущен, то выполнение продолжится выполнением следующего оператора **case**. Утверждения **case** можно рассматривать как метки. Выполнение будет продолжаться до тех пор, пока встретится утверждение **break** или до окончания утверждения **case**.

Нижеприведенный пример иллюстрирует процесс выбора из меню:

```
switch(ch) {
  case 'e':
    enter( );
    break:
  case 'l':
    list( );
    break;
  case 's':
    sort( );
    break;
  case 'q':
    exit(0);
  default:
    cout << "Unknown Command\n";
}
```

template

Ключевое слово **template** используется для создания параметризованных функций и классов. (Их также называют "функции-шаблоны" и "классы-шаблоны".) При создании шаблона функции или класса тип данных, над которыми они оперируют, задаются одним или несколькими типами-шаблонами, которые автоматически заменяются фактическими типами данных при создании конкретного экземпляра функции или класса.

Общая форма функции-шаблона показана ниже:

```
template <class Ttype> ret-type func-name(parameter list)
{
  // body of function
}
```

Здесь *Ttype* представляет собой шаблон для имени типа данных, используемых функцией. Это имя может использоваться в пределах определения функции. Однако, оно является только шаблоном, и компилятор автоматически заменит его нужным типом данных при создании конкретного экземпляра функции. С помощью утверждения **template** можно определить несколько параметризованных типов данных, указывая их в списке с использованием запятой в качестве разделителя.

Общая форма параметризованного класса приведена ниже:

```
template <class Ttype> class class-name {
    // body of class
}
```

Здесь *Ttype* представляет собой имя-шаблон, которое компилятор при построении конкретного экземпляра класса заменит реальным типом данных. При необходимости вы можете определить несколько параметризованных типов данных, используя список с запятой в качестве разделителя. Функции-члены параметризованного класса также автоматически становятся параметризованными.

this

this представляет собой указатель, автоматически передаваемый функциям-членам при их вызове. Этот указатель ссылается на объект, вызвавший функцию.

throw

throw представляет собой часть механизма обработки исключений C++ и используется для отбрасывания исключений. Обработка исключений в C++ построена на трех ключевых словах: **try, catch** и **throw**. Говоря наиболее общими терминами, утверждения программы, которые требуется наблюдать на предмет исключений, содержатся в блоке **try**. Если исключение (то есть, ошибка) имеет место, она отбрасывается (с помощью **throw**). Исключения отлавливаются с помощью **catch** и обрабатываются. Нижеприведенное обсуждение освещает эту тему более подробно.

Как уже упоминалось, каждое утверждение, которое отбрасывает исключение, должно выполняться в пределах блока **try**. (Функции, вызываемые из блока **try**, также могут отбрасывать исключения.) Любое исключение можно отловить с помощью утверждения **catch**, следующего немедленно за утверждением **try**. Общая форма утверждений **try** и **catch** приведена ниже:

```
try {
  // try block
}
catch (type1 arg) {
  // catch block
}
catch (type2 arg) {
  // catch block
}
catch (type3 arg) {
  // catch block
}
.
.
.
catch (typeN arg) {
  // catch block
}
```

Блок **try** должен содержать ту часть вашей программы, которую требуется наблюдать на предмет обработки исключений. Она может быть очень короткой (например, несколько утверждений в пределах одной функции) или всеохватывающей и включающей в себя абсолютно все, даже функцию **main()**, что приведет к наблюдению за всей программой.

При отбрасывании исключения оно отлавливается соответствующим утверждением **catch**, которое и обрабатывает его. С утверждением **try** может ассоциироваться несколько утверждений **catch.** Какое из них фактически будет использовано, определяется типом отброшенного исключения. Это означает, например, что если тип данных, определенный утверждением **catch**, совпадает с типом данных исключения, то исполняться будет именно это утверждение **catch** в обход остальных. Когда исключение отлавливается, arg получает свое значение. Отлавливать можно любые типы данных, включая создаваемые вами классы. Если отбрасывания исключений не происходит (это значит, что не происходит ошибок в пределах блока **try**), то не выполняется и ни одно из утверждений **catch**.

Общая форма утверждения **throw** приведена ниже:

```
throw exception;
```

Утверждение **throw** должно исполняться из блока **try** или из любой функции, вызванной (прямо или косвенно) из этого блока. *Exception* представляет собой передаваемое значение.

Если вы отбросите исключение, для которого нет соответствующего утверждения **catch**, это может привести к аварийному завершению программы.

true

true представляет собой одно из двух значений, которые может принимать булевская переменная.

try

try представляет собой часть механизма обработки исключений в C++. Подробную информацию можно найти в пояснению к утверждению **throw**.

typedef

Утверждение **typedef** позволяет создавать новые имена для существующих типов данных. Это утверждение имеет следующую общую форму:

typedef t*ype-specifier new-name*;

Например, для того, чтобы слово **balance** можно было использовать вместо **float**, необходимо включить в программу следующую строку:

```
typedef float balance;
```

typeid

Утверждение **typeid** получает информацию о типе данных объекта. Оно имеет следующую общую форму:

typeid(*object*)

Здесь *object* представляет собой объект, тип которого вы получаете. **Typeid** возвращает ссылку на объект типа **type_info**, которая описывает тип объекта, заданного параметром *object*. Для использования typeid необходимо включить в программу заголовочный файл TYPEINFO.H.

Поскольку определение **typeid** происходит во время выполнения программы, это утверждение применимо к полиморфичным типам. Это значит, что его можно использовать для определения типа объекта, на который указывает указатель базового класса применительно к полиморфичным классам.

Класс **type_info** определяет следующие общедоступные члены:

bool operator==(const type_info *&ob*) const;
bool operator!=(const type_info *&ob*) const;
bool before(const type_info *&ob*) const;
const char *name( ) const;

Перегрузка == и != обеспечивает все необходимое для сравнения типов. Функция **before()** возвращает значение true, если вызывающий объект расположен перед объектов, используемым в качестве параметра. (Эта функция в основном предназначена для внутреннего использования. Ее возвращаемое значение не имеет ничего общего с наследованием и иерархией классов.) Функция **name()** возвращает указатель на имя типа.

typename

typename задает имя, которое в декларации шаблона обозначает имя типа.

union

Ключевое слово **union** используется для присвоения одной или нескольким переменным одной и той же области памяти. Это означает, что все члены объединения (**union**) совместно используют одну и ту же область памяти. Общая форма утверждения **union** приведена ниже:

union *union-name* {
  type *member1*;
  type *member2*;
  .
  .
  .
  type *memberN*;
} *object-list;*

Формально, объединение создает тип класса. Доступ к членам осуществляется с помощью операторов (.) и (->).

unsigned

Утверждение **unsigned** определяет тип беззнаковых целых. Например, для того, чтобы объявить переменную **big** как беззнаковую целую, необходимо включить в программу следующую декларацию:

```
unsigned int big;
```

using

См. **namespace**.

virtual

virtual декларирует виртуальные функции. Виртуальные функции представляют собой специальные функции-члены, которые декларируются в пределах базового класса, а затем над ними приобретают приоритет функции производных классов. При вызове виртуальной функции будет выполняться та версия этой функции, которая фактически выполняется во время выполнения программы.

void

Спецификатор типа **void** используется в основном для того, чтобы декларировать функции, не возвращающие значений.

volatile

Модификатор типа **volatile** сообщает компилятору о том, что содержимое переменной может изменяться методами, которые не были явно определены. Примером могут служить переменные, значение которых изменяется аппаратно (таймеры, прерывания и т. п).

wchar_t

wchar_t задает тип 16-битных символов. 16-битные символы также широко известны под названием "широких" (*wide*). Такие типы могут содержать символьные наборы всех человеческих языков.

while

Цикл **while** имеет следующую общую форму:

```
while(expression) {
  statements
}
```

Если в теле цикла имеется единственное утверждение или объект, то фигурные скобки не обязательны и их можно опустить. Цикл **while** совершает итерации до тех пор, пока контрольное выражение expression имеет значение true. Таким образом, если значение этого выражения изначально ложно, цикл не выполнит ни одной итерации. Ниже приведен пример, выводящий на экран числа от 1 до 100:

```
t = 1;

while(t<=100) {
  cout << t << endl;
  t++;
}
```

Index

B

BASIC 319

C

Const_cast 171

D

Dynamic_cast 171

F

FIFO 63

I

Internet 318, 369

J

Java 318, 369
- аналоги с C++ 370
- байт-код 371
- отличия от C++ 372
- пакеты 385
- суперкласс 385
- финализаторы 381

L

LIFO 63
Long 14

M

Modula-2 287

N

NPOS 189

S

Sentinels 32
SmallBASIC 319
Static_cast 171
String 182

T

Template 48

V

Volatile 172

W

Windows95 181, 387
WindowsNT 387
WWW 369

А

Алгоритм n-квадратичный 24
Аллокатор 185

Б

Бинарное дерево 47, 79
  как метод реализации разреженных массивов 125

В

Ветвь 79
Виртуальная память 126
Виртуальные функции 159
Выражения, синтаксическая проверка 114

К

Класс
  string 182
  встроенный, type_info 159
  полиморфичный 158
  родовой 46
  стандартный
    C++ 182
    string 184
Ключевое слово
  dynamic_cast 159
  template 13
Компилятор
  Borland C++ 182
  Microsoft C++ 182

Конструктор копирования 294
Контейнерные классы 46

М

Массив
ограниченный 47, 48
разреженный 125
на базе массива указателей 142
физический 127
Метод
быстрой сортировки 37
поиска 42
Множества
объединение 288
пересечение 288
пустое 288
разность 288

Н

Надмножество 288

О

Оператор
перегрузки 38
преобразования типов C++ 171
присваивания 107
Операторы унарные 104
Операция со стеком pop 57
Очередь 47
циклическая 54

П

Параметризованные функции, способы построения 15
Побитовое копирование 294
Подмножество 288
Поиск
бинарный 42, 43
ключ 87
метод 42
последовательный 42
сортировка, параметризованные функции 14
стандартные библиотечные функции 42
Программы-калькуляторы, разработка 105

Р

Разбор выражений
задача 90

работа с переменными 105
рекурсивно-нисходящий алгоритм 93
Разрушающее извлечение данных 63
Рекурсивные функции 80

С

Сжатие данных 208
универсальные методы 232
Синтаксические ошибки, обработка 115
Сортировка
библиотечные функции 14
пузырьковый метод 22
Список
с двойными связями, перестройка поврежденного 63
связный 47, 174
при реализации разреженных массивов 125
Способы прохождения дерева 80
Стек 47
Строкоориентированный редактор, разработка 198

У

Удаление объекта из списка с двойными ссылками 69

Х

Хэширование
список коллизий 147
функция 147

Ч

Числовые выражения, правила построения 91

Ш

Шаблоны 13
Шифрование
алгоритм 211
метод пакетной перестановки 224

Э

Электронные таблицы 127
Эйлера константа 28

Я

Язык программирования
C 160
FORTRAN 160
Java 318, 369
Pascal, 160

Готовятся к выпуску в издательстве "BHV — Санкт-Петербург":

В серии "... в подлиннике":

Каждая книга из данной серии — это всегда энциклопедичность, полнота и достоверность информации. Эти книги представляют собой блестящие руководства по работе с конкретными аппаратными и программными средствами.

Книги из серии ***"... в подлиннике"*** представляют несомненный интерес для тех пользователей, которые хотят добиться профессионализма в своей работе.

1. Р. Персон. Excel для Windows 95 в подлиннике
2. Р. Винтер, П. Винтер. Microsoft Office для Windows 95 в подлиннике
3. Р. Персон. Access для Windows 95 в подлиннике
4. Д. Мэтчо. Delphi 2 в подлиннике
5. Р. Уоллес. PageMaker 6 для Windows 95 в подлиннике

В серии "мастер":

Книги этой серии написаны экспертами в области разработки и эксплуатации сложных программных и аппаратных комплексов. Только в книгах этой серии вы найдете систематическое и полное изложение специальных аспектов современных профессиональных компьютерных технологий и сможете выбрать оптимальные и эффективные решения проблем системного и прикладного программирования.

Серия ***"мастер"*** предлагает незаменимый инструмент в работе программиста, дает уникальный шанс расширить границы ваших профессиональных возможностей, передает вам опыт и знания специалистов наивысшей квалификации.

1. У. Мюррей, К. Паппас. Программирование для Windows в различных средах
2. А. Мешков, Ю. Тихомиров. Visual C++ и MFC
3. Ч. Петзолд. Программирование в Windows 95
4. У. Мюррей, К. Паппас. Visual C++
5. А. Мавричева. Создание WWW-страниц

В серии "... одним взглядом":

Книги этой серии являются незаменимыми для ежедневной работы справочниками, позволяющими быстро ориентироваться в сложных программных и аппаратных средствах вычислительной техники. Наглядность и компактность размещения больших объемов информации — отличительная особенность книг серии ***"... одним взглядом"***. Специально разработанная

іетодология представления информации позволяет мгновенно отыскать іолную информацию по интересующему вас вопросу.

Иатериал каждой из книг, входящих в серию, организован таким образом, ітобы одним взглядом можно было охватить функционально законченную 'ему, относящуюся к тому или иному режиму работы (например, "Работа с 'аблицами", "Фильтры" и т. д.). С этой целью на левой странице каждого раз-юрота этих книг представлены диалоговые окна и меню. Кнопки, команды, ›пции и сообщения имеют номера, в соответствии с которыми на правой 'транице даются наименование (с переводом) элемента и его краткое описа-іие. Нужную вам тему можно найти по оглавлению или по предметному ука-ателю.

. В. Матвеев, Ю.Тихомиров. MS Visual FoxPro одним взглядом
!. С. Молотков. Программирование в MS Visual FoxPro одним взглядом
ǀ. С. Пономаренко. Adobe PageMaker 6 одним взглядом

3 серии "... в примерах":

Неискушенный читатель, делающий первые шаги в освоении вычисли-'ельной техники, найдет все необходимые для своей работы сведения в книгах, представленных в серии ***"... в примерах"***. Эти книги отличает про-'тота и доступность изложения материала, и вместе с тем полнота, позво-іяющая познакомиться на практике со всеми возможностями того или іного программного продукта. Книги этой серии содержат большое коли-іество наглядных примеров.

l. С. Пономаренко. Adobe PageMaker 6 в примерах
!. С. Пономаренко. CorelDRAW! 6 в примерах
ǀ. П. Дарахвелидзе, Е. Марков. Delphi 2 в примерах
I. А. Хомоненко. Word 7.0 для Windows 95 в примерах
5. Ю. Бекаревич, Н.Пушкина. Access для Windows 95 в примерах
5. К. Максимов и др. Netscape 2.0 в примерах

Сделать заказ или получить более подробную информацию можно в издательстве:

тел: (812) 541 8551; факс: (812) 541 8461
e-mail: bhv@mail.nevalink.ru; root@bhv.spb.su

и в библиотеке издательства:

http://www.bhv.ru

Книги издательства "BHV—Санкт-Петербург" в продаже:

Автор, название книги	Количество страниц
Бестселлеры	
1. "Ресурсы Windows NT"	720
2. "Сетевые средства Windows NT"	496
3. М. Пайк. "Internet в подлиннике":	640
Серия "...в подлиннике"	
1. Ф. Новиков, А. Яценко. "Microsoft Office в целом" Возможна поставка дискеты по цене 6000 руб.	336
2. Б. Лоренс. "Novell NetWare 4.1 в подлиннике"	720
3. К. Айден. "Аппаратные средства PC"	544
4. Р. Персон. "Word для Windows 95 в подлиннике"	704
5. Р. Персон. "Windows 95 в подлиннике"	736
Серия "...в примерах"	
1. "От Windows 3.1 к Windows 95 за один день"	96
2. М. Фролов. "Мультимедиа в примерах"	128
3. С. Пономаренко. "Adobe Photoshop 3.0 в примерах"	320
4. М. Нольден. "Ваш первый выход в Internet"	240
Серия "...одним взглядом"	
1. А. Старшинин. "MS PowerPoint одним взглядом"	112
2. С. Пономаренко. "Adobe Photoshop 3.0 одним взглядом"	160
3. С. Пономаренко. "CorelDRAW! 5.0 одним взглядом"	144
4. А. Хомоненко. "MS Word для Windows 95 одним взглядом"	176
5. И. Серебрянский. "Novell NetWare 4.1 одним взглядом"	160
6.С. Пономаренко. "CorelDRAW! 6 для Windows 95 одним взглядом"	176
Новинки	
1. Р. Тидроу. "Управление реестром Windows 95"	288
2. Е. Вострокнутов. "MS Excel для Windows 95 одним взглядом"	144
3. С. Карпенко, И. Шишигин. "Internet в вопросах и ответах"	464
4. В. Байков. "Internet в примерах"	220
5. К. Айвинс. "OS/2 в вопросах и ответах"	300
6. Ю. Тихомиров, А. Мешков. "MS Access для Windows 95 одним взглядом"	196
7. С. Пономаренко. "Adobe PageMaker 6.0 для Windows 95 одним взглядом"	144

Книги издательства "BHV—Санкт-Петербург" можно приобрести в магазинах:

	Санкт-Петербург	
1. "Дом книги"	Невский пр., 28	(812) 312 0184
2. "Техническая книга"	ул. Пушкинская, 2	(812) 164 6565
3. Магазин № 55	П.С., Большой пр., 34	(812) 230 9966
4. "Шанс на Садовой"	ул. Садовая, 40	(812) 315 3117
5. "Энергия"	Московский пр., 189	(812) 293 0147
6. "Эллит" (СПб. ЭТУ)	ул. Попова, 5	(812) 234 8987
7. "Академ. книга"	Литейный пр., 57	(812) 230 1328
8. "Питер"	ул. Пионерская, 22	(812) 230 2596
9. "Родина"	Ленинский пр., 127	(812) 254 2104
10. "Веком"	пр. Славы, 15	(812) 109 0391
	Москва	
1. "Дом тех. книги"	Ленинский пр., 40	(095) 137 6019
2. "Библио-Глобус"	ул. Мясницкая, 6	(095) 928 8744
3. "Молодая Гвардия"	ул. Б. Полянка, 28	(095) 238 0032
4. "Центр-Техника"	ул. Петровка, 15	(095) 924 3624
5. ТД "Москва"	ул. Тверская, 8	(095) 229 7355
6. "Дом книги"	ул. Новый Арбат, 8	(095) 203 8242
7. "Дом педагогической книги"	ул. Большая Дмитровка, 7/5	(095) 229 4392
8. "Мир"	ул. Ленинградский пр., 78	(095) 152 8282
9. "Новый"	ш. Энтузиастов, 24	(095) 362 0923
10. "РиС"	ул. Красного маяка, 11	(095) 313 8345
11. "Ридас"	Новоданиловская наб., 9	(095) 954 3044
12. "Кнорус"	Милютинский пер., 19/4	(095) 928 6269
	Нижний Новгород	
1. "Дом книги"	ул. Советская,14	(8312) 44 2273
2. "Знание"	пр. Ленина, 3	(8312) 42 6589
	Екатеринбург	
1. "Дом книги"	ул. Антона Валика,12	(3432) 59 4200
2. "Техническая книга"	ул. К. Либкнехта, 16	(3432) 51 1664
3. Магазин №14	ул. Челюскинцев, 23	(3432) 53 2490
	Владивосток	
1. Приморский ТД книги	ул. Светланская, 43	(4232) 23 8212
	Уфа	
1. "Азия"	ул. Гоголя, 62	(3472) 22 5662
	Каунас	
1. "Technine literatura"	Кэстуче,17, фирма "Смалтия"	(0127) 22 4576
	Новосибирск	
1. ТОО "Эмбер"	ул. Спартака, 16	(3832) 69 3650
2. ТОО "ГПНТБ-Поиск"	ул. Восход, 15	(3832) 66 8567
	Красноярск	
1. Магазин "Факел"	ул. Мичурина, 23	(3912) 33 7450

Книги серии "мастер" написаны экспертами в области разработки и эксплуатации программных и аппаратных комплексов. Только в книгах этой серии Вы найдете систематическое и полное изложение специальных аспектов современных профессиональных компьютерных технологий и сможете выбрать оптимальные и эффективные решения проблем прикладного программирования. Серия "мастер" предлагает незаменимый инструмент в работе программиста, дает уникальный шанс расширить границы Ваших профессиональных возможностей, передает Вам опыт и знания специалистов наивысшей квалификации.

Для опытных программистов и всех, кто готов к самостоятельным исследованиям:

- Профессиональные советы и подходы, помогающие создавать надежные и эффективные программы
- Разработка и реализация языковых интерпретаторов и компиляторов
- Практические примеры построения иерархий классов
- Интеграция модулей ассемблерных кодов в программы на C++
- Расширение среды C++ путем добавления новых типов данных (класс String _ новый стандартный класс ANSI C++)
- Информация о новых возможностях C++, RTTI, обработка исключений и многое другое
- Краткий обзор языка Java, его отличие от C++

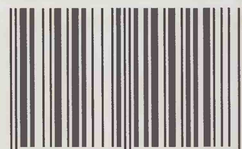
Разработка языка C++ позволила создавать программное обеспечение нового поколения. Языки программирования и операционные системы создаются, развиваются и устаревают, а C++ по-прежнему остается мощным и современным средством программирования. Если Вы хотите писать профессиональные программы _ эта книга для Вас! Книга уникальна по своему содержанию, в ней излагается широкий круг специальных вопросов, которые в комплексе не рассмотрены ни в одном подобном издании.

Г. Шилдт

Теория и практика C++

МАСТЕР

РУКОВОДСТВО ДЛЯ ПРОФЕССИОНАЛОВ


ISBN 5-7791-0029-2
9 785779 100298